

GEOSPATIAL DATA SCIENCE ESSENTIALS

101 Practical
Python Tips and
Tricks

MILAN JANOSOV

Geospatial Data Science Essentials

101 Practical Python Tips and Tricks

Written by

Milan Janosov

2024

Copyright © 2024 by Milan Janosov

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed “Attention: Permissions Coordinator,” at the address below.

Disclaimer:

The author has made every effort to ensure the accuracy and completeness of the information contained in this book. However, the author assumes no responsibility for errors, omissions, or damages resulting from the use of the information herein. The content is provided "as is" and is subject to change without notice.

AI Disclaimer:

This book was authored solely by Milan Janosov. However, various AI tools, including language models, were utilized to assist in the drafting, editing, and proofreading processes. The insights and information provided by these AI tools were reviewed and curated by the author to ensure accuracy, relevance, and quality. The author maintains full responsibility for the content and its presentation.

First Edition: 2024

Purpose

Creating a starting point for this book was truly a challenge, even after spending countless days and nights rewriting and revising section after section. So, cutting it short, my goal was simply to compile the essentials of geospatial data science in Python that I wish someone had handed me when I began working with spatial data.

This book avoids overburdening details and theoretical depths — there are great books on that already. Instead, I wanted to focus on the most practical aspects and tons of Python coding on geospatial data science topics. My aim was to show you how to use Python for geospatial analytics, to provide a solid foundation, and to offer an overview of the different tools and methods of the current geospatial data science stack. Given the rapid growth of this field, I did not even attempt to make this a perfect and completely exhaustive guide; instead, it is a collection of essential tips and tricks, bits and pieces of knowledge to get you started on your next geospatial project and hop on this awesome emerging branch of data analytics.

While you are most certainly reading a book now, initially, I did not even want to write one. I simply set out to create a cheat sheet, but as the content expanded rapidly, I realized it deserved better packaging than just one or two social media posts. Hence, it turned into an introductory book on geospatial science. To get there, I was particularly moved by the support from my community on social media and in my courses alike. The countless sparkling discussions and learning materials inspired me to properly work out the book format and make this into something quick to

process, practical, and ready for immediate use. I hope this book achieves that goal, and only time and you, dear reader, will tell how successful I have been with that.

Why Now?

I'm not a salesman, and I have no intention of trying to sell you on geospatial data science. However, I do have two compelling data points that highlight why now is an excellent time to dive into and master this field. According to [Markets and](#) the geospatial analytics market is projected to reach nearly USD 150 billion during the next few years. Additionally, a recent piece on GIS Geography, ["1000 GIS Applications &](#) lists literally a thousand possible applications for GIS. While these examples are not exhaustive quantitative research, they clearly illustrate the magnitude and versatility of spatial analytics from the financials to the domains of application.

So, why now? When I shared my purpose for writing this book, I mentioned that I wished someone had given me these essentials many years ago. This book has been long overdue, also sitting on my to-do list for over a year. So now, I am making up for this, finally bringing it to life and sharing the foundational knowledge of geospatial data science with you.

What is in This Book?

This is a technical book that covers ten topics and altogether 101 smaller sections, each dedicated to common geospatial analytical tasks. It assumes some beginner-level experience in data analysis, statistics, and math.

However, I intended this book to be introductory and act as an appetizer to the field, so you don't need years of experience to benefit from it.

While you can read the entire outline in the Table of Contents, I also want to explain the logic behind the structure. First, we will learn about geometries as data structures, which show us how to connect geographic locations to data records stored in tables. Then, we will integrate this knowledge into the Pythonic data analysis framework provided by Pandas and GeoPandas, including geospatial data visualization — also known as drawing maps. Next, we will cover map projection, spatial indexing, and geocoding — topics you may not have encountered before, but we cover these from the very beginning.

After exploring vector data, we will dedicate an entire section to raster data, which ranges from satellite images to the cover of this book. Then, we will explore the widely used crowdsourced geospatial data platform, OpenStreetMap. Finally, we will study two more advanced topics: spatial networks and geospatial statistics combined with machine learning. This way, I aim to cover the most typical areas, although this list, like all lists, will never be complete.

Two final notes on the structure of the chapters: each chapter contains numbered sections, starting with Section 1 in the first chapter and concluding with Section 101 at the end of the last chapter. Each section includes a short theoretical introduction and an explanation of the code blocks that follow, usually complemented by visuals and written interpretation. While all Python packages used are referenced at their first appearance, I also collected all the technical details, environment, and library versions in the Appendix at the end of the book.

What is Not in This Book?

While the 101 sections covered are certainly comprehensive, they are far from complete. Frankly, it was neither my intention to cover every possible angle nor a task I think is realistically feasible, given how fast this field is growing. Although the coverage is fairly extensive, a few topics, despite their importance and frequent use in my daily practice, are intentionally left out.

I also wanted to highlight a few of these to help you orient yourself to the content of this book. Specifically, this book does not discuss any GIS applications like QGIS and ArcGIS, big data analytics frameworks such as Dask and Spark, or databases ranging from SQL to DuckDB. It also excludes web-based tools like Mapbox, behind-the-scenes powerhouse libraries such as GDAL, and the expansive and rapidly evolving field of generative AI. My goal was to keep the focus sharp and manageable, allowing a deep dive into the essentials without an overwhelming breadth. Everything else not listed in the outline is also beyond the scope of this book.

Is This Book for You?

Yes, this book is for you if you are:

A data scientist or analyst interested in location-related topics.

A GIS expert planning to learn about data science.

Someone who loves maps, Python programming, or both.

Why Me?

This is the part where I introduce myself, so hi there! I was born in Budapest, Hungary, in 1991. With a background in physics and biophysics, I earned my PhD in network and data science in 2020. I studied and researched at the Eötvös Loránd University and the Central European University in Budapest, at the Barabási Lab in Boston, and the Bell Labs in Cambridge. I was selected for the Forbes 30u30 list in 2020, while Data Science Connect put me on their 99 Data Influencers to Follow in 2023 list. I am a regular columnist at Towards Data Science and the Data Visualization Society's Nightingale. My first book, titled DATA, debuted with OpenBooks in Hungarian in 2023.

I am the founder of Geospatial Data Consulting, former co-founder of Datapolis and former chief data scientist at Baoba. Besides, I was a research affiliate at the Central European University, a research expert at the European Commission, and a senior data scientist at Maven7. In each role, my responsibility was to bring network science and spatial analytics to life, from data-driven urban planning to geo-relevant insurtech products and worldwide policy-making.

My work has been featured in academic and public media venues alike, such as Nature Social Science Research, GQ, Times Higher Education, New Scientist, New York Times, TechXplore, The Economic Times, Fossbytes, Futurism, The Times, Phys.org, Gamestar, and more.

How to Reach Me

There are multiple ways to reach me and stay updated with my latest work, research, and tutorials. You can follow:

My daily social media posts on [X \(formerly](#) and

My newsletter, Milan's Data Science Insights, on [Substack](#) and

My tutorials, Milan's Data Stories, on and

or just visit my website,

What You Get

Along with purchasing this book, you get the following:

An ebook in .epub format, available via Amazon.

A nicely formatted PDF, which you can download from

The original Jupyter notebooks containing all the code for each chapter, accessible

A requirements.txt file, which contains a list of all the libraries and their versions used

AI Disclaimer

While this book was definitely my idea, I also used several AI tools to make this happen. In particular, I used ChatGPT for brainstorming and editing and Grammarly for language checking, as I am not a native speaker. This may raise a few questions, which I have answered here. If you have any more questions, just reach out to me:

Could I have written a similar quality work without using AI tools?

Most likely, yes.

Would it have taken much longer to finish this book without using AI tools?

Yes, they most definitely sped up the process.

Did I rely solely on AI tools for the content?

No, the content is based on my expertise and research, with AI tools used to enhance and expedite the process.

Did AI tools replace my expertise and judgment?

No, AI tools complemented my expertise, but the final content and decisions were my own.

Were there any parts of the book where AI tools were particularly helpful?

Yes, especially in brainstorming, organizing content, commenting code, and refining language.

Is there a single prompt for creating this book?

No, instead there are hundreds of prompts often rerun a dozen times to edit standalone pieces.

Did I review and edit all AI-generated content?

Yes, all AI-generated content was thoroughly reviewed, edited, and corrected if needed.

Without further ado, let's get into it and start learning the geospatial essentials through 101 practical Python tips and tricks!

Happy Learning,
Milan Janosov

Getting started

Here I would like to give you some pointers on how to get started with this book, and cover some of my personal best-practices.

Programming environment

I have been using [Jupyter Notebook](#) for data analytics for many years, combined with

Environment installation

There are multiple parallel ways to install the relevant environment and libraries; here, I show you one way I have been practicing for many years, both on Linux and Mac. If you would rather hear it than read it, check out my [YouTube video](#) titled “Setting up a Python Environment” - find my [channel](#) here.

First, you will need a command line terminal, where you run the following command to create a virtual environment, for instance, called GEO. While OSMnx is not the typical Numbetr 1 geospatial package that comes to mind (even if it is probably my favorite), it has a really nice and complete installer, which has been the fastest way for me so far to get a new environment up and running for spatial analytics:

```
conda create -n testgeo4 -c conda-forge --strict-channel-priority osmnx  
jupyter
```

After installation and saying yes to any possible question the installer has, let's active the environment and print the Python version it uses:

```
conda activate testgeo4  
python --version
```

In my system, it is currently Python

Library installation overview

In my experience, properly installing all geo-related dependencies can be time-consuming and painful, that's why I especially love the OSMNx installer. Once its done, we are usually out of the woods, and the remaining librareies are easy enough to install even within a Jupyter notebook (remember, after installing a package within the notebook, you should restart the kernel).

This is how you can quickly install a library, for instance, `session_info`, which tells us the exact versions of packages within our work environment:

In

```
import sys
-m pip install session_info
```

In

```
import session_info
```

The `session_info` package quickly confirms the base setup I have, along with some basic system info:

session_info 1.0.0

IPython 8.26.0

jupyter_client 8.6.2

jupyter_core 5.7.2

jupyterlab 4.2.3

notebook 7.2.1

Python 3.12.4 | packaged by conda-forge | (main, Jun 17 2024,
10:11:10) [Clang 16.0.6]

macOS-14.3.1-x86_64-i386-64bit

Session information updated at 2024-07-16 18:00

Check and install all libraries

Below, I will provide you with the full setup guide, split into code cells package-by-package. This section has been tested in multiple scenarios. However, you still may encounter some initial installation errors. Hopefully not, but if so - let me know!

First, let's review the packages we are going to install. This is a quick list, while each library will be discussed and referenced in more detail throughout the coming chapters.

A collection of perceptually uniform colormaps for data visualization.

A library for adding background maps to geospatial visualizations in Python.

A graphics pipeline system for creating meaningful representations of large datasets quickly.

Exploratory Spatial Data Analysis (ESDA) tools for spatial statistics.

A library for creating interactive maps using Leaflet.js and Python.

An open-source project that makes working with geospatial data in Python easier by extending pandas.

A Python library for geocoding, reverse geocoding, and obtaining geographical coordinates.

A library for working with H3, a hexagonal hierarchical spatial index.

The core library of PySAL, providing foundational data structures and algorithms for spatial analysis.

A comprehensive library for creating static, animated, and interactive visualizations in Python.

A collection of utility toolkits that extend the capabilities of Matplotlib. V

A library for the creation, manipulation, and study of complex networks.

A fundamental package for scientific computing with Python, providing support for arrays and matrices.

A Python package to work with street networks and other spatial data from OpenStreetMap.

A powerful data manipulation and analysis library for Python.

A graphing library that makes interactive, publication-quality graphs online.

A WebGL-powered framework for visual exploratory data analysis of large datasets.

A module that implements pseudo-random number generators for various distributions.

A library for reading and writing geospatial raster data.

A library that extends xarray for raster data.

A library that provides spatial indexing to accelerate spatial queries.

A Python library used for scientific and technical computing.

A library for the manipulation and analysis of planar geometric objects.

A library for machine learning, providing simple and efficient tools for data analysis.

A library for spatial econometrics, part of PySAL.

A library for statistical modeling and testing.

A library for working with labeled multi-dimensional arrays.

A rich architecture for interactive computing.

A package for managing Jupyter protocol clients and servers.

Provides core functionality for Jupyter projects.

The Jupyter interactive notebook.

A library for Python session information.

Core libraries

First, let's check the core of our installation, which started with OSMNx:

In

```
import osmnx as ox
```

Another central library we will use is GeoPandas. While GeoPandas came out with a new release lately; here, we are still using one of the previous versions due to historical reasons. The next line ensures that we have the right version installed:

In

```
-m pip install  
import geopandas as gpd
```

Built-in libraries

Let's make sure we have the following core analytical libraries in place:

In

```
import numpy as np
import scipy
import random
import pandas as pd
import matplotlib.pyplot as plt
import mpl_toolkits
```

Additional library installations

Throughout this book, we will use several geospatial analysis-specific libraries. During the following code blocks, we install them all and test them by quickly importing each of them. Where we need specific versions of a library, it is marked as we did with GeoPandas. By running all these cells, you should have a workspace ready to process this entire book.

In

```
-m pip install colorcet
import colorcet
```

In

-m pip install

import contextily

In

-m pip install datashader

import datashader

In

-m pip install esda

import esda

In

-m pip install folium

import folium

In

-m pip install geopy

import geopy

In

-m pip install h3

import h3

In

-m pip install libpysal

import libpysal

In

-m pip install networkx

import networkx as nx

In

-m pip install rtree

import rtree

In

-m pip install plotly

import plotly.express as px

In

```
-m pip install pydeck  
import pydeck as pdk
```

In

```
-m pip install rasterio  
import rasterio
```

In

```
-m pip install rioarray  
import rioarray
```

In

```
-m pip install shapely  
import shapely
```

In

```
-m pip install scikit-learn  
import sklearn
```

In

```
-m pip install spreg
```

```
import spreg
```

In

```
-m pip install xarray
```

```
import xarray
```

In

```
-m pip install statsmodels
```

```
import statsmodels.api as sm
```

Final requirements

Finally, let's create a standard file containing all dependencies based on the complete installation guide.

In

The final session info matching all libraries and versions I used while developing this book:

| | |
|--------------|--------|
| colorcet | 3.1.0 |
| contextily | 1.1.0 |
| datashader | 0.16.3 |
| esda | 2.5.1 |
| folium | 0.17.0 |
| geopandas | 0.14.4 |
| geopy | 2.4.1 |
| h3 | 3.7.7 |
| libpysal | 4.11.0 |
| matplotlib | 3.9.1 |
| mpl_toolkits | NA |
| networkx | 3.3 |
| numpy | 2.0.0 |
| osmnx | 1.9.3 |
| pandas | 2.2.2 |
| plotly | 5.22.0 |
| pydeck | 0.9.1 |

| | |
|-----------|--------|
| rasterio | 1.3.10 |
| rioxarray | 0.16.0 |
| rtree | 1.3.0 |

| | |
|--------------|----------|
| scipy | 1.14.0 |
| session_info | 1.0.0 |
| shapely | 2.0.5 |
| sklearn | 1.5.1 |
| spreg | 1.5.0 |
| statsmodels | 0.14.2 |
| xarray | 2024.6.0 |

Click to view modules imported as dependencies

| | |
|----------------|--------|
| IPython | 8.26.0 |
| jupyter_client | 8.6.2 |
| jupyter_core | 5.7.2 |
| jupyterlab | 4.2.3 |
| notebook | 7.2.1 |

Python 3.12.4 | packaged by conda-forge | (main, Jun 17 2024,
10:11:10) [Clang 16.0.6]
macOS-14.3.1-x86_64-i386-64bit

And exporting these into a "requirements.txt" file:

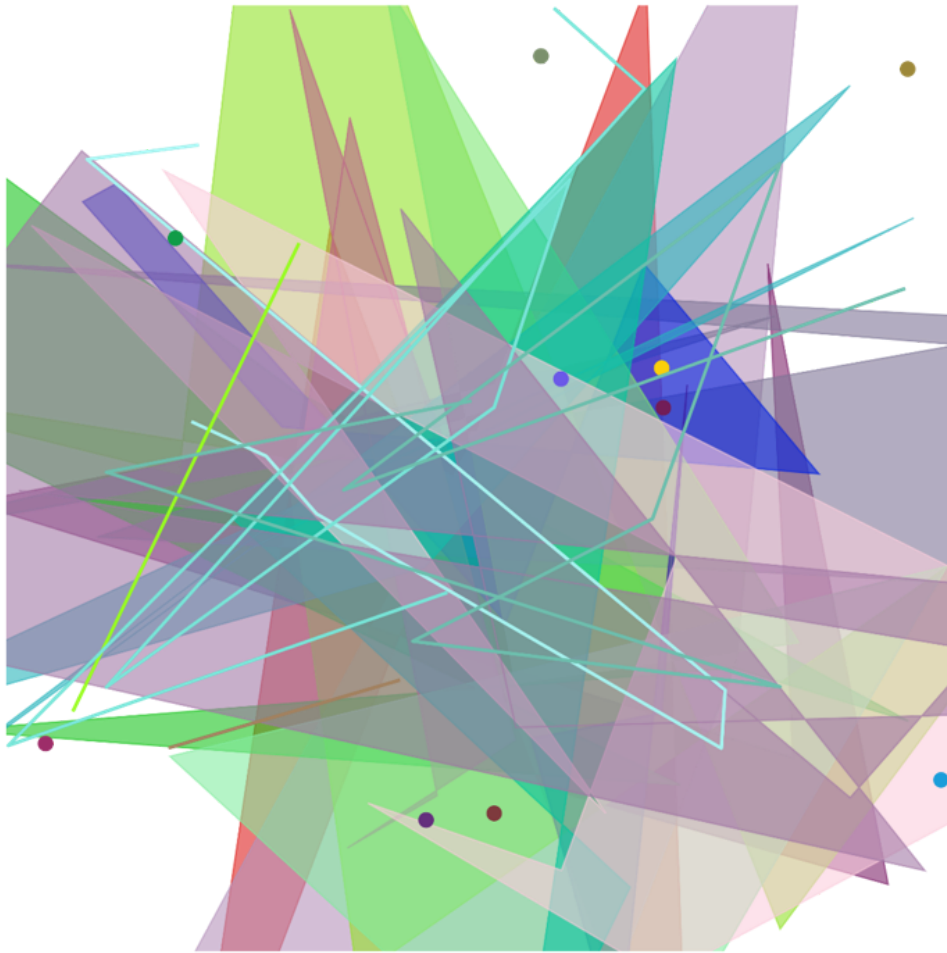
In

```
# Extract the relevant section from session_info output
session_data =
pattern = r'(\S+)\s+(\d+\.\d+\.\d+)'
```

```
matches = session_data)
# Create a list of strings in the format "package==version"
requirements = [f"{match[0]}=={match[1]}" for match in matches if
match[1] != 'NA']

# Write to requirements.txt
with open("requirements.txt", "w") as file:
for requirement in requirements:
```

Introduction to Geometries



Geometries are the cornerstone of spatial data, acting as the geographical framework that links various pieces of information to specific locations on the Earth's surface. They provide the spatial structure necessary to conduct a wide array of analyses in geospatial data science. Therefore, understanding the theory and mastering the implementation of geometries is crucial for effectively working with any geospatial data.

In this chapter, we will begin by exploring the basic types of geometries, such as points, lines, and polygons. These geometries can represent a diverse range of geographical entities, from the locations of bars and restaurants to street segments, riverbeds, country borders, and crop fields. Following our exploration of basic geometry types, we will review essential operations for transforming and combining these geometries. Learning these concepts will enable us to perform advanced spatial analyses and create insightful visualizations.

To achieve this, we will utilize the powerful [Shapely](#) library, specifically designed for creating, manipulating, and analyzing geometric objects.

1. Creating a Point

A point is the simplest type of geometric object, representing a single location in a two-dimensional space. It is defined by a pair of coordinates (x, y) and is often used to represent discrete places such as cities, landmarks, or any specific location of interest. In geospatial data science, points are fundamental for mapping and spatial analysis. Oftentimes, such a point location is called a point of interest

Shapely provides a straightforward way to create point geometries using the Point constructor. Let's see how by creating a point placed at the (1, 1) point, printing the point object, and then visualizing it by outputting the point by the code cell.

In

```
1 # Import the Point constructor
2 from shapely.geometry import Point
3
4 # Create a point with coordinates (1, 1)
5 point = Point(1, 1)
6
7 # Print the point
8 print("The type of the point: ", type(point), "\n")
9
10 # Print the point
11 print("The point itself: ", point, "\n")
12
13 # Show the point on the cell's output
```

14 point

The type of the point: `'shapely.geometry.point.Point'`>

The point itself: `POINT (1 1)`

Out



As this cell shows, we indeed created a point, which is visualized by a small circle in the Jupyter environment.

2. Creating Line Segments

A line segment is a geometric object that represents a series of connected points in a two-dimensional space. It is defined by a sequence of coordinate pairs (x, y), consisting of multiple straight line segments.

In Shapely, lines are stored in the `LineString` data structure and are usually used to represent one-dimensional spatial objects such as roads, rivers, and sidewalks. In geospatial data science, `LineStrings` are essential for modeling and analyzing linear spatial features.

Shapely provides a straightforward way to create and manipulate `LineString` geometries using the `LineString` constructor, which we will see in the following cell where we create a line segment connecting the (0, 0), (1, 1), and (3, 4) points. Once creating the object, we print it and visualize it on the cell output.

In

```
1 # Import the LineString constructor
2 from shapely.geometry import LineString
3
4 # Create a line from two points (0, 0), (1, 1), and (3, 4)
5 line = LineString([(0, 0), (1, 1), (3, 4)])
6
7 # Print the line
8
9 print("The type of the line: ", type(line), "\n")
10
```

```
11 # Print the point
12 print("The line itself: ", line, "\n")
13
14 # Print the line
15 print(line, "\n")
16
17 # Show the line on the cell's output
18 line
```

The type of the line: 'shapely.geometry.linestring.LineString'>

The line itself: LINESTRING (0 0, 1 1, 3 4)

LINESTRING (0 0, 1 1, 3 4)

Out



Here, we created a `LineString`, which consists of two straight segments, as shown on the cell output.

3. Creating Polygons and Multipolygons

A polygon is a two-dimensional geometric object that represents a closed shape defined by a series of connected points, where the first and last points are identical to close the shape. Polygons are used to represent areas such as countries, lakes, or any other bounded region. In geospatial data science, polygons are fundamental for mapping and analyzing spatial areas. Later, we will use Shapely's Polygon constructor to create polygon objects.

One step further, we will meet a which is a geometric object that represents a collection of multiple polygons treated as a single entity. This is useful for representing spatial features consisting of several disconnected areas, such as multiple land parcels. Shapely provides a straightforward way to create and manipulate multipolygon geometries using the MultiPolygon constructor.

Similar to the generalization from polygon to multipolygon, Shapely also has implemented MultiPoint and MultiLineString classes, which are collections of multiple points and LineStrings, respectively, treated as single entities. Finally, GeometryCollections are the most general collections that can contain a mix of different geometries, such as points, LineStrings, and polygons, all within a single object. These collections are useful for complex spatial analyses involving multiple types of geometries.

As mentioned earlier, Shapely provides a straightforward way to create and manipulate both polygon and multipolygon geometries using the Polygon and MultiPolygon constructors, which we will explore in the following. First, we create a polygon called `polygon_1` - a triangle with the vertices

placed at the (0, 0), (1, 1), and (1, 0). Second, we create another triangle with its corners placed at the (1, 0), (1, 1), and (1.5, 0) points. Third, we create a MultiPolygon by joining these two triangles. After each geometry creation step, we also print the geometries and visualize them at the cell's outputs.

In

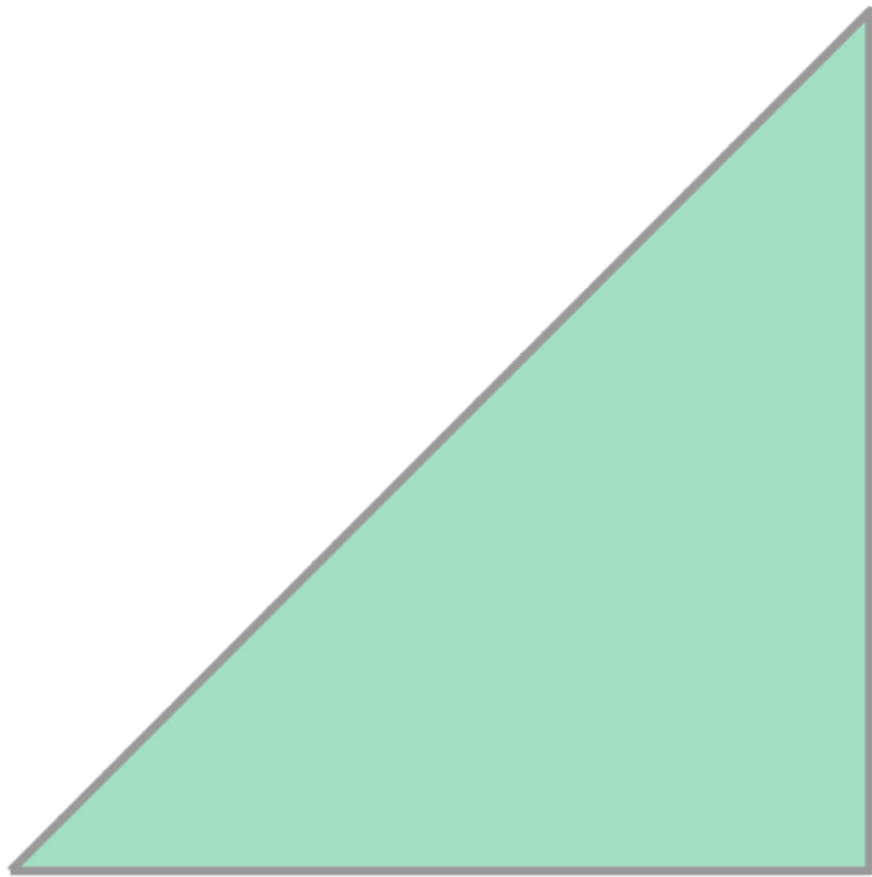
```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon, MultiPolygon
3
4 # Create the first polygon with a series of points
5 polygon_1 = Polygon([(0, 0), (1, 1), (1, 0)])
6
7 # Print the first polygon
8 print("The type of the first polygon: ", type(polygon_1), "\n")
9
10 # Print the point
11 print("The first polygon itself: ", polygon_1, "\n")
12
13 # Print and show the first polygon
14 print(polygon_1, "\n")
15 polygon_1
```

The type of the first polygon: 'shapely.geometry.polygon.Polygon'>

The first polygon itself: POLYGON ((0 0, 1 1, 1 0, 0 0))

POLYGON ((0 0, 1 1, 1 0, 0 0))

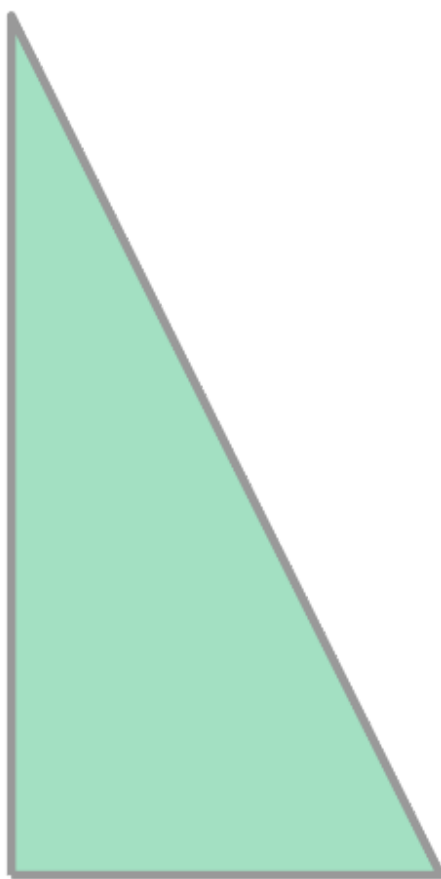
Out



In

```
1 # Create the second polygon with another series of points
2 polygon_2 = Polygon([(1, 0), (1, 1), (1.5, 0)])
3 polygon_2
```

Out



In

```
1 # Import the MultiPolygon constructor from Shapely
```

```
2 from shapely.geometry import MultiPolygon
3
4 # Combine the two polygons into a multipolygon
5 multipolygon = MultiPolygon([polygon_1, polygon_2])
6
7 # Print the multipolygon
8 print("The type of the multipolygon: ", type(multipolygon), "\n")
9
10 # Print the point

11 print("The first multipolygon itself: ", multipolygon, "\n")
12
13 # Print and show the multipolygon
14 print(multipolygon, "\n")
15
16 multipolygon
```

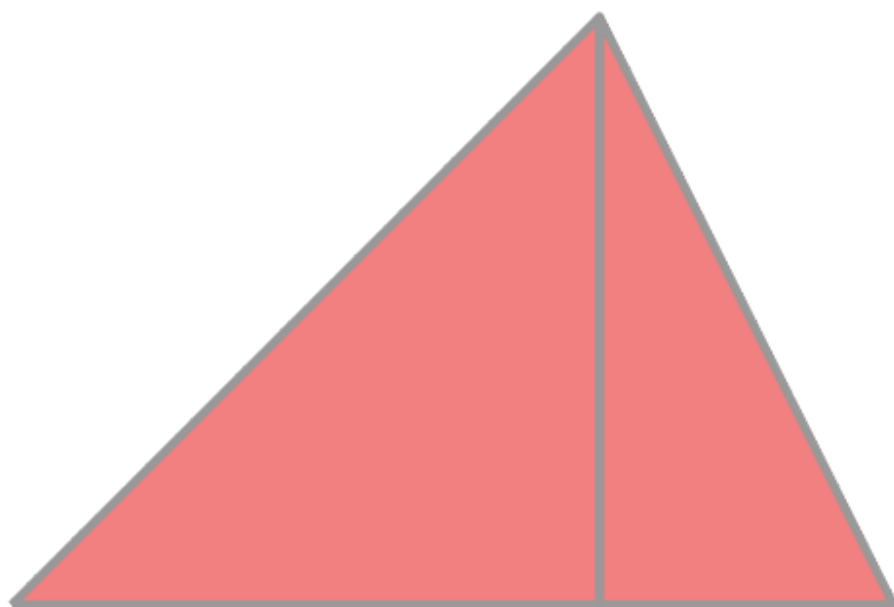
The type of the multipolygon:

'shapely.geometry.multipolygon.MultiPolygon'>

The first multipolygon itself: MULTIPOLYGON (((0 0, 1 1, 1 0, 0 0)), ((1 0, 1 1, 1.5 0, 1 0)))

MULTIPOLYGON (((0 0, 1 1, 1 0, 0 0)), ((1 0, 1 1, 1.5 0, 1 0)))

Out



4. Buffering a Geometry

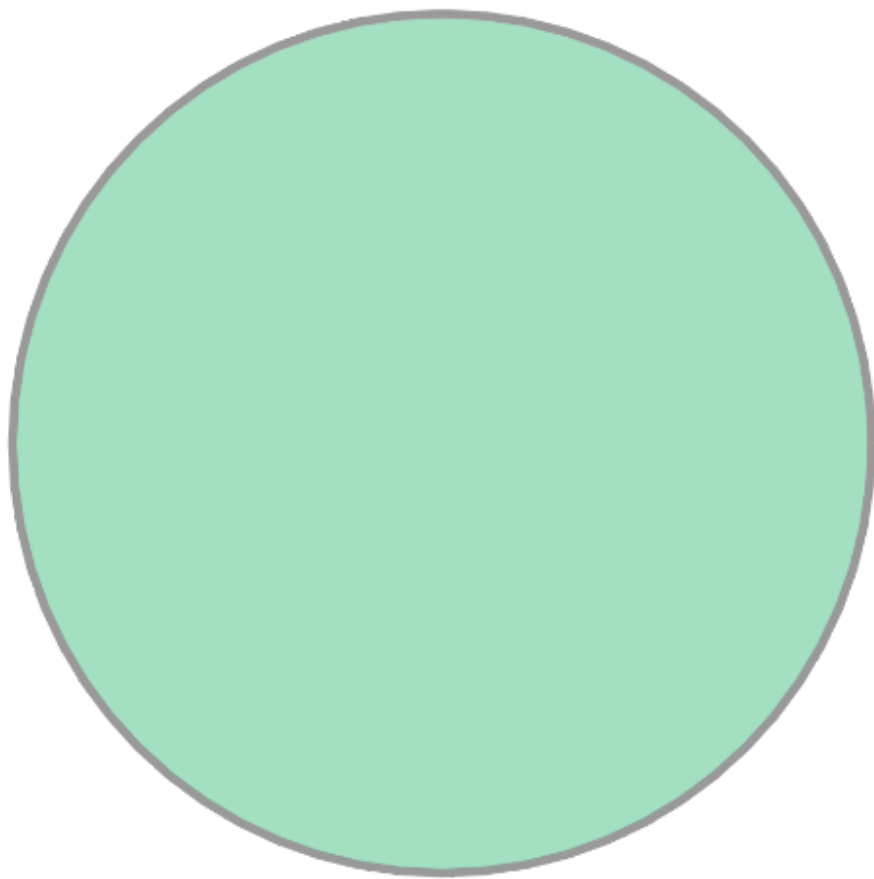
Buffering means transforming an existing geometry into a new geometry that represents all points within a specified distance from the original geometry. This operation is useful for creating zones around spatial features, such as protective zones around natural resources, areas of influence around infrastructure, or any region that needs to be considered within a certain distance from a given place. Hence, we can buffer points, lines, and polygons as well. All buffering operations result in a polygon.

Shapely provides a straightforward way to buffer geometries using the `buffer` method, which can be applied to any geometric object. The following example shows how we can buffer a point at (0, 0) with a buffer radius of 1 unit, which will result in a circle. Then, we output the circle using the usual methods.

In

```
1 # Import the Point constructor from Shapely
2 from shapely.geometry import Point
3
4 # Create a point at coordinates (0, 0)
5 point = Point(0, 0)
6
7 # Buffer the point by a specified distance (e.g. 1 unit) to create a circle
8 circle =
9
10 # Show the buffered geometry (circle)
11 circle
```

Out



Now, we repeat the same buffering steps on a triangle, resulting in a buffered polygon, a triangle with cornered vertices.

In

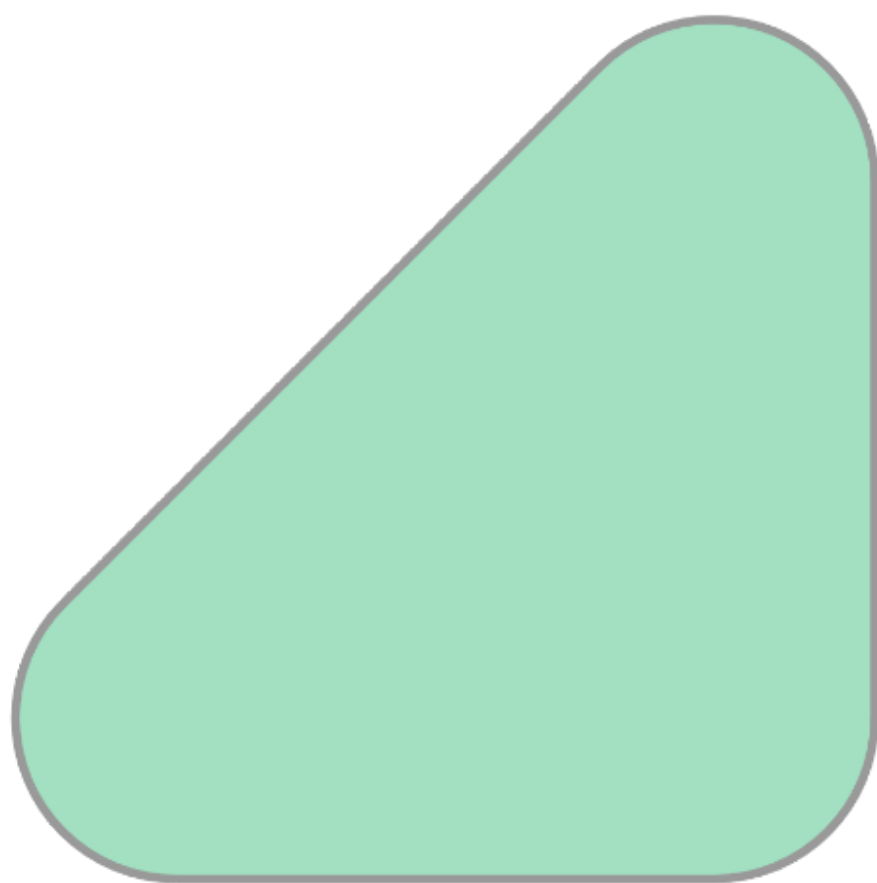
```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon
3
4 # Create a polygon from a series of points
5 # (the first and last points must be the same to close the shape)
6 polygon = Polygon([(0, 0), (1, 1), (1, 0)])
7
8 # Buffer the polygon by a distance of 0.3 units
9 buffered_polygon =

```

10

```
11 # Show the buffered geometry
12 buffered_polygon
```

Out



5. Set Operations on Geometries

Set operations on geometries allow us to quantify spatial relationships and perform operations between different geometric shapes. These operations are fundamental in spatial analysis and are widely used in countless domains of spatial analytics.

Shapely provides a range of set operations that can be performed on geometries, including intersection, union, difference, and symmetric difference. Now we review the most frequently used ones.

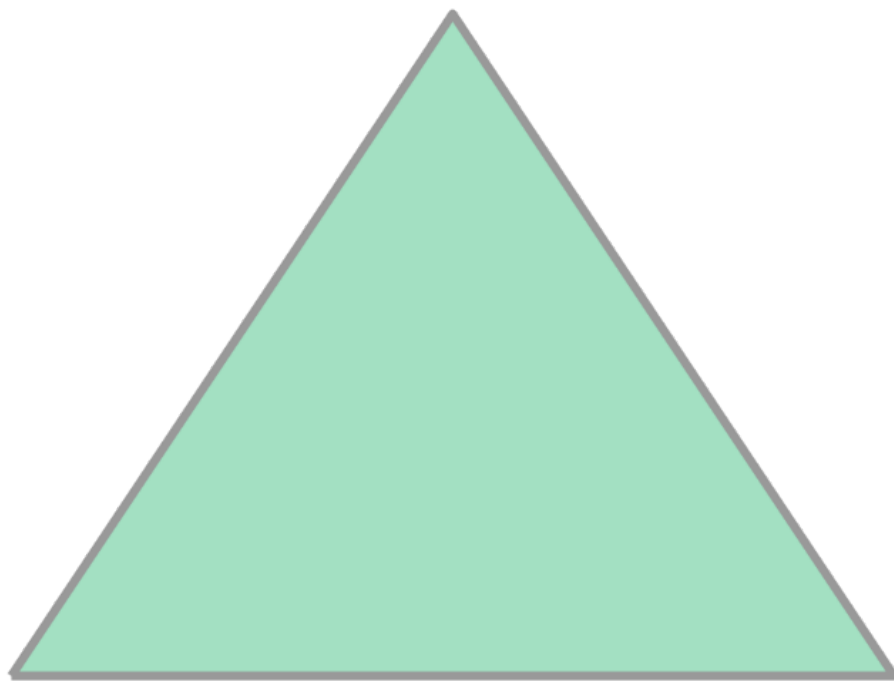
Intersection

The intersection operation finds the common area shared by two geometries. This is useful for determining overlapping regions between spatial features - geometric objects representing real-world entities in a spatial context. To test the intersection method, in the following two cells, we create a triangle and a circle using the previously introduced steps. Then, we create their intersection and output the resulting polygon.

In

```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon
3
4 # Create the first polygon
5 polygon1 = Polygon([(0, 0), (2, 0), (1, 1.5)])
6 polygon1
```

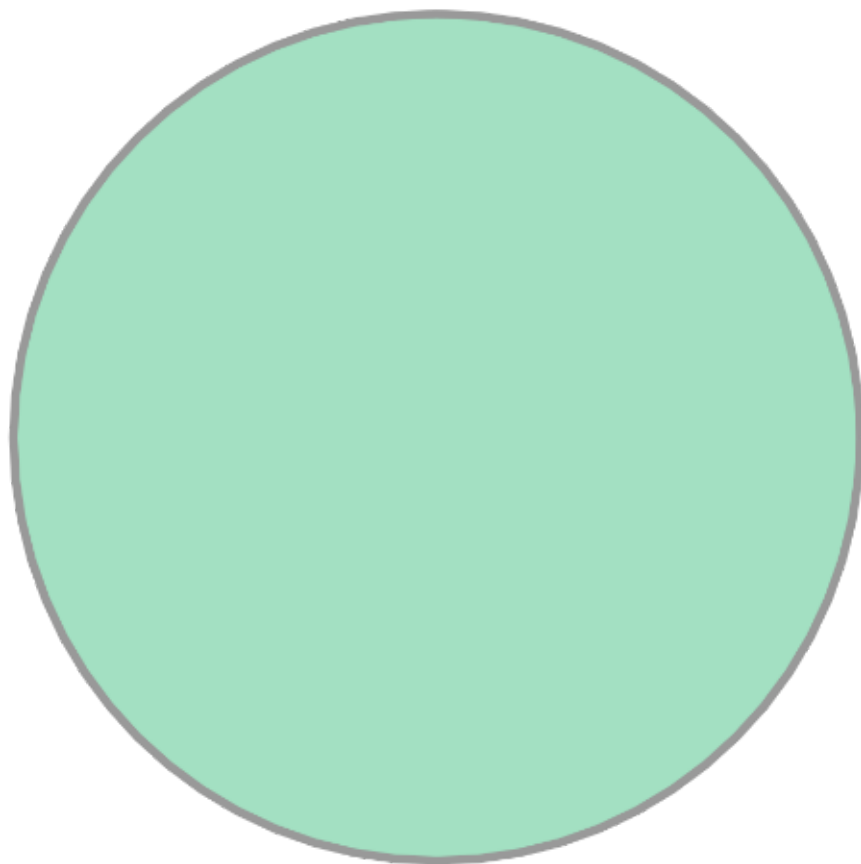
Out



In

```
1 # Create the second polygon
2 polygon2 =
3 polygon2
```

Out



In

1 # Compute the intersection of the two polygons
2 intersection =

```
3
4 # Print and show the intersected geometry
5 print(f'Intersection: {intersection}', "\n")
6
7 intersection
```

```
Intersection: POLYGON ((0.9230119102873612 1.3845178654310417,
0.9238795325112865 1.3826834323650905, 0.9569403357322088
1.2902846772544625, 0.9807852804032303 1.1950903220161286,
0.9951847266721969 1.0980171403295604, 1 1, 0.9951847266721969
0.9019828596704393, 0.9807852804032304 0.8049096779838718,
0.9569403357322088 0.7097153227455377, 0.9238795325112867
0.6173165676349102, 0.881921264348355 0.5286032631740023,
0.8314696123025452 0.4444297669803978, 0.773010453362737
0.3656067158363545, 0.7071067811865476 0.2928932188134525,
0.6343932841636456 0.2269895466372631, 0.5555702330196023
0.1685303876974549, 0.4713967368259978 0.1180787356516451,
0.3826834323650898 0.0761204674887133, 0.2902846772544623
0.0430596642677911, 0.1950903220161283 0.0192147195967696,
0.0980171403295608 0.0048152733278032, 0.0000000000000001 0, 0 0,
0.9230119102873612 1.3845178654310417))
```

Out



Union

The union operation combines two geometries into a single geometry that covers all areas of the input geometries. This is useful for merging adjacent areas or combining multiple features into a single entity. As this example shows, we can, for instance, easily join the previously created triangle and circle to turn them into one merged polygon containing all areas covered by either of these geometries.

In

1 # Compute the union of the two polygons

2 union =

3

4 # Show the union geometry

5 union

Out



Difference

The difference operation subtracts the area of one geometry from another, resulting in a geometry that represents the part of the first geometry that does not overlap with the second geometry. This is useful for finding the area of a feature that lies outside another feature. Similarly to the union command, we can test the difference method by subtracting the second polygon from the first and then doing it the other way around. Calculating the difference in both

directions demonstrates that the order of the components matters when performing a difference operation.

In

1 # Compute the difference of polygon1 and polygon2 (polygon1 minus polygon2)

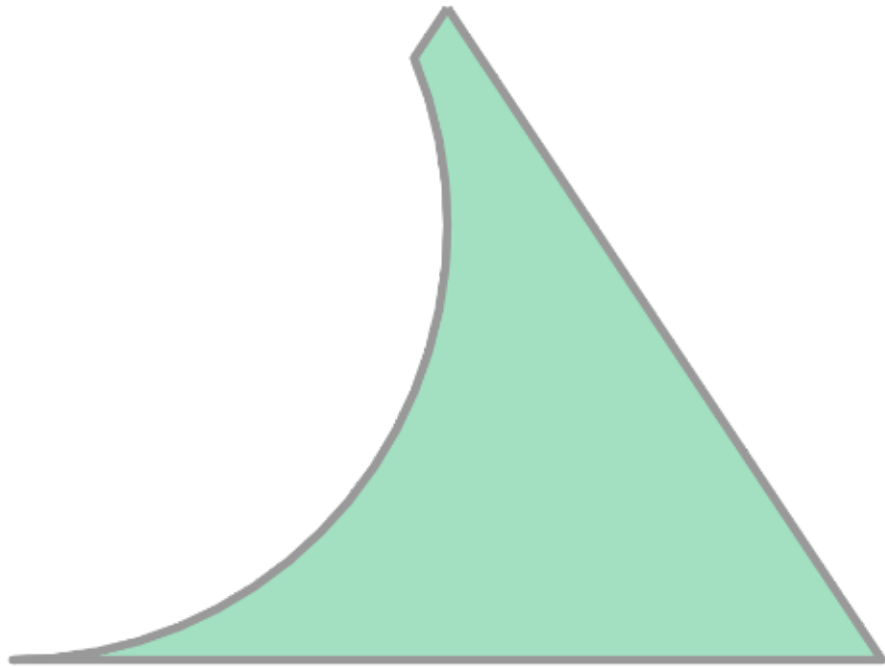
2 difference =

3

4 # Show the difference geometry

5 difference

Out



In

1 # Compute the difference of polygon2 and polygon1 (polygon2 minus polygon1)

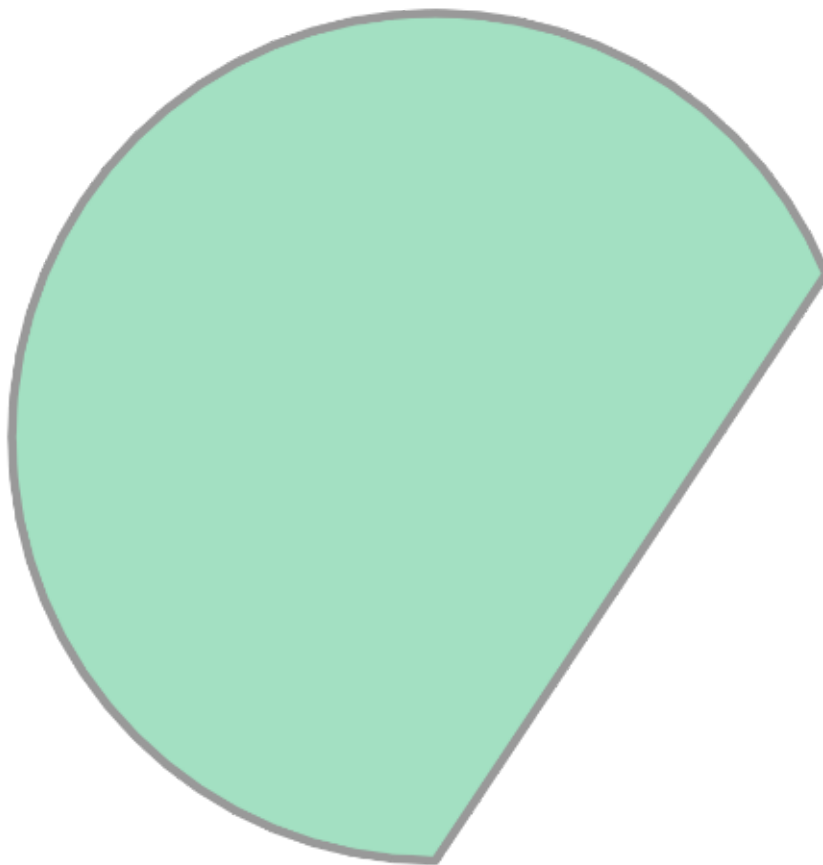
2 difference =

3

4 # Show the difference geometry

5 difference

Out



Symmetric Difference

The `symmetric_difference` operation finds the areas that are in either of the input geometries but not in their intersection. This results in a geometry that

represents the non-overlapping areas of both geometries. Similarly to the union and command difference, we can test the symmetric difference as follows.

In

```
1 # Compute the symmetric difference of the two polygons
2 symmetric_difference =
3
4 # Show the symmetric difference geometry
5 symmetric_difference
```

Out



6. Area and Perimeter Computation

Shapely provides robust built-in methods to compute geometric properties such as the area and circumference (or perimeter) of various shapes. These properties are crucial in many spatial analyses, such as land use planning, environmental impact assessments, and natural resource management.

Area Computation

The area of a geometry is a measure of the surface it covers. Shapely can compute the area of two-dimensional objects - Polygons and MultiPolygons. The area is computed in the coordinate system of the geometry, which typically represents square units of the spatial reference system used. Later, we will discuss the different coordinate systems in a dedicated chapter.

To test the area computation, first, we create two triangles and compute and print their areas. Then, we join them into a MultiPolygon and repeat these analytical steps. Finally, we also create their union, creating a single polygon containing both these triangles. Again, we repeat the same analytical steps.

In

```
1 # Import the Polygon and MultiPolygon constructors from Shapely
2 from shapely.geometry import Polygon, MultiPolygon
3
4 # Create individual polygons
5 polygon1 = Polygon([(0, 0), (1, 1), (1, 0)])
6 polygon2 = Polygon([(1, 0), (1, 3), (3, 0)])
7
```

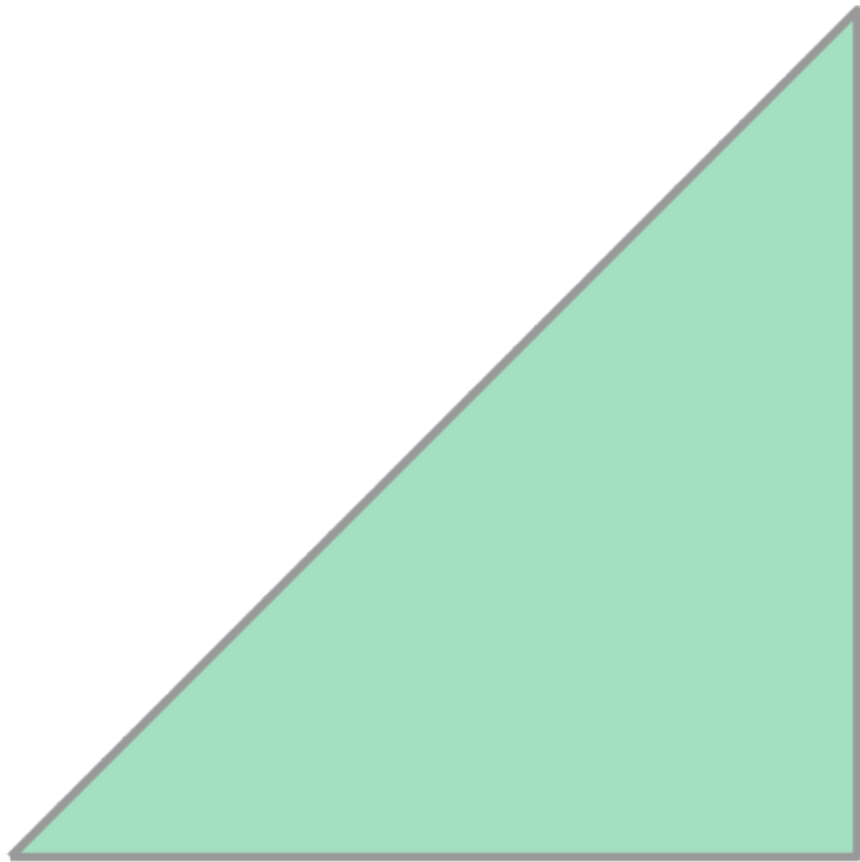
```
8 # Compute the area of the polygons
9 area_polygon1 =

10 area_polygon2 =
11
12 # Print the area
13 print(f'Area of the polygon1: {area_polygon1}', "\n")
14 print(f'Area of the polygon1: {area_polygon2}', "\n")
15
16 polygon1
```

Area of the polygon1: 0.5

Area of the polygon1: 3.0

Out



In

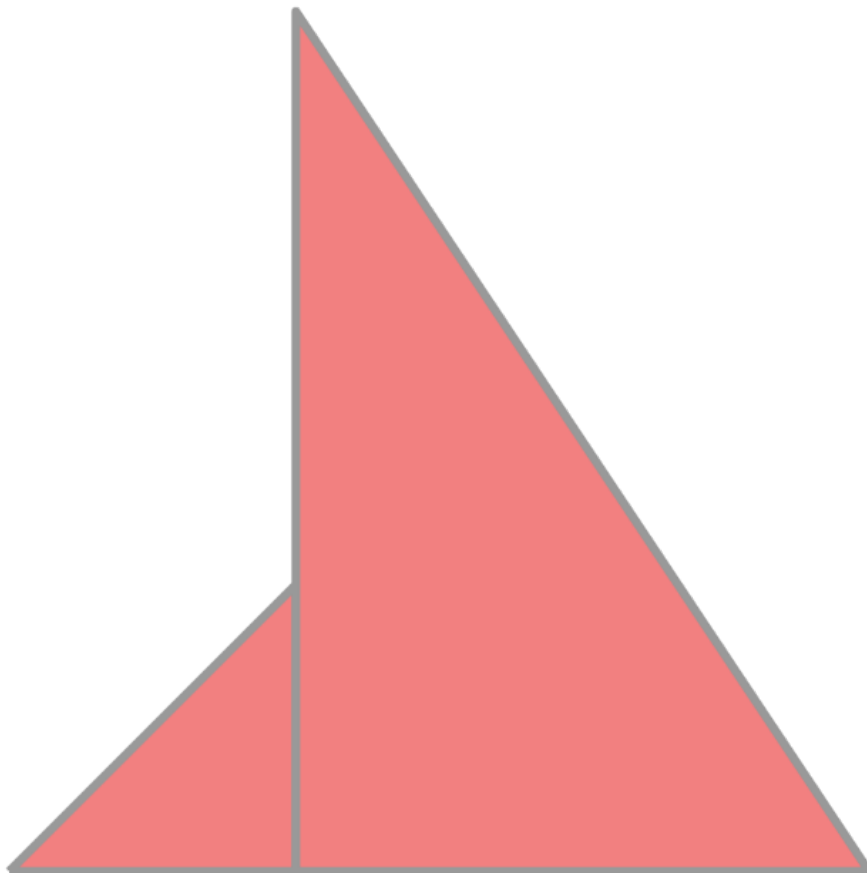
```
1 # Combine polygons into a multipolygon
2 multipolygon = MultiPolygon([polygon1, polygon2])
3
4 # Compute the area of the multipolygon
5 area_multipolygon =
6
7 # Print the area
```



```
8 print(f"Area of the multipolygon: {area_multipolygon}", "\n")
9
10 multipolygon
```

Area of the multipolygon: 3.5

Out

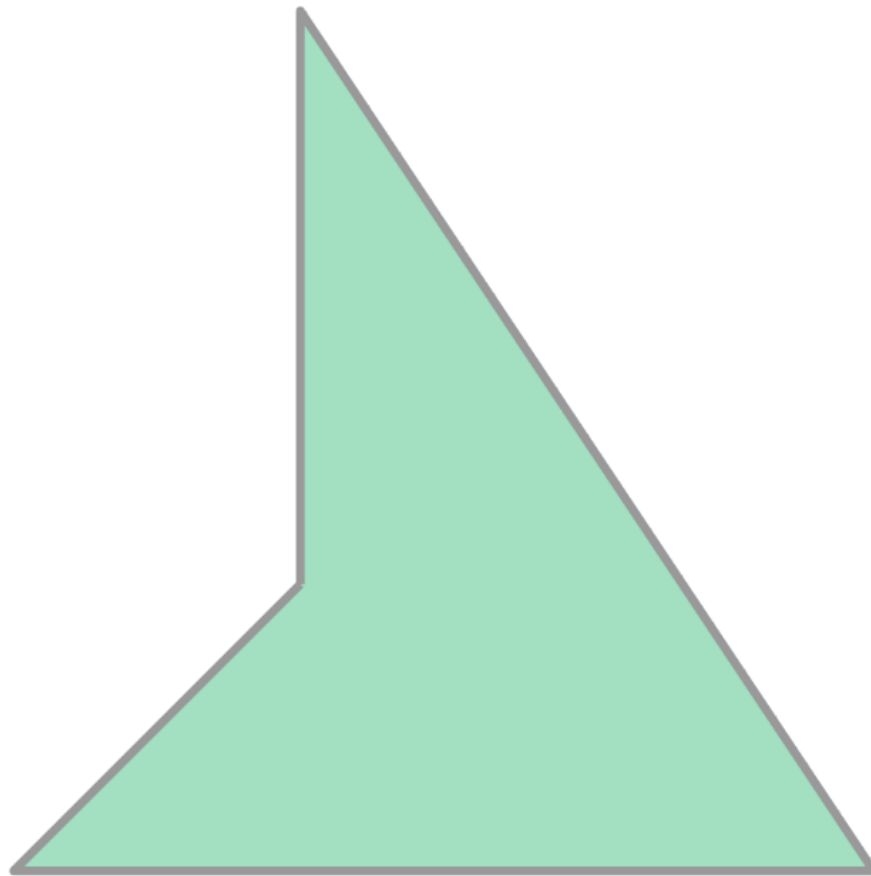


In

```
1 # Combine polygons by taking their union
2 polygon =
3
4 # Compute the area of the multipolygon
5 area_polygon =
6
7 # Print the area
8 print(f"Area of the polygon: {area_polygon}", "\n")
9
10 polygon
```

Area of the polygon: 3.5

Out



As the two triangles only touch each other (so they don't overlap), both the multi polygon built off from them and their union have an area that is equal to the area of the triangles separately, measuring $0.5 + 3 = 3.5$ square units total.

Circumference (Perimeter) Computation

The circumference, or perimeter, of a geometry is the total length of its boundary lines. Shapely can compute the perimeter of one- and two-dimensional geometries, such as LineStrings, Polygons, and MultiPolygons.

Now, we follow the previous area computations and first compute the perimeter of the two triangles as separate polygons, then as a MultiPolygon, and finally, as a unified Polygon.

In

```
1 # Compute the perimeter of the polygons
2 perimeter_polygon1 =
3 perimeter_polygon2 =
4
5 # Print the perimeter of the polygons
6 print(f'Perimeter of polygon1: {perimeter_polygon1}', "\n")
7 print(f'Perimeter of polygon2: {perimeter_polygon2}', "\n")
8 print(f'Combined perimeter: {perimeter_polygon1 +
perimeter_polygon2}', "\n")
```

Perimeter of polygon1: 3.414213562373095

Perimeter of polygon2: 8.60555127546399

Combined perimeter: 12.019764837837084

In

```
1 # Compute the perimeter of the multipolygon
2 perimeter_multipolygon =
3
```

```
4 # Print the perimeter of the multipolygon
5 print(f'Perimeter of the multipolygon: {perimeter_multipolygon}', "\n")
```

Perimeter of the multipolygon: 12.019764837837084

In

```
1 # Compute the perimeter of the joint polygon
2 perimeter_polygon =
3
4 # Print the perimeter of the joint polygon
5 print(f'Perimeter of the unified polygon: {perimeter_polygon}', "\n")
```

Perimeter of the unified polygon: 10.019764837837085

Here, we can see again that the multipolygon's perimeter is the same as the sum of the individual polygons' perimeters. However, when we take the union, the touching edge of the two triangles will be dissolved. Hence, the resulting polygon will have a shorter perimeter than the two individual polygons combined with the union operation.

7. Computing Centroids

The centroid of a geometry is the geometric center or the "center of mass" of that shape. For simple geometries like points, lines, and polygons, the centroid can be thought of as the average position of all the points in the geometry. Hence, a point is its own centroid, while a line's centroid is its middle point.

Let's compute the centroid of three different geometries and see how a point at (0, 0) turns into itself, how we get the center point of a line connecting (0, 0) and (0, 1), and how we get the center point for the triangle of (0, 0), (1, 1), and (1, 0).

In

```
1 # Import the Point constructor from Shapely
2 from shapely.geometry import Point
3
4 # Create a point
5 point = Point(0, 0)
6
7 # Compute the centroid of the point
8 centroid =
9
10 # Print and show the centroid
11 print(f'Centroid of the point: {centroid}', "\n")
12 centroid
```

Centroid of the point: POINT (0 0)

Out



In

```
1 # Import the LineString constructor from Shapely
2 from shapely.geometry import LineString
3
4 # Create a line
5 line = LineString([(0, 0), (0, 1)])
6
7 # Compute the centroid of the line
8 centroid =
9
10 # Print and show the centroid
11 print(f'Centroid of the line: {centroid}', "\n")

12 centroid
```

Centroid of the line: POINT (0 0.5)

Out



In

```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon
3
4 # Create a polygon from a series of points
5 polygon = Polygon([(0, 0), (1, 1), (1, 0)])
6
7 # Compute the centroid of the polygon
```

8 centroid =

9

10 # Print and show the centroid

11 print(f'Centroid of the polygon: {centroid}', "\n")

12 centroid

Centroid of the polygon: POINT (0.6666666666666666
0.3333333333333333)

Out



8. Enclosing Polygons

Enclosing polygons are geometries that completely contain another geometry and represent simplified or generalized versions of the original shape.

Shapely provides two primary methods for creating enclosing polygons from existing geometries: the `envelope` and the `convex_hull` methods.

Envelope

The envelope of a geometry is the smallest rectangle (aligned with the coordinate axes) that completely contains the geometry. In this example, we create a concave polygon and first draw its enclosing envelope, which, as expected, is a rectangle covering the polygon in each direction.

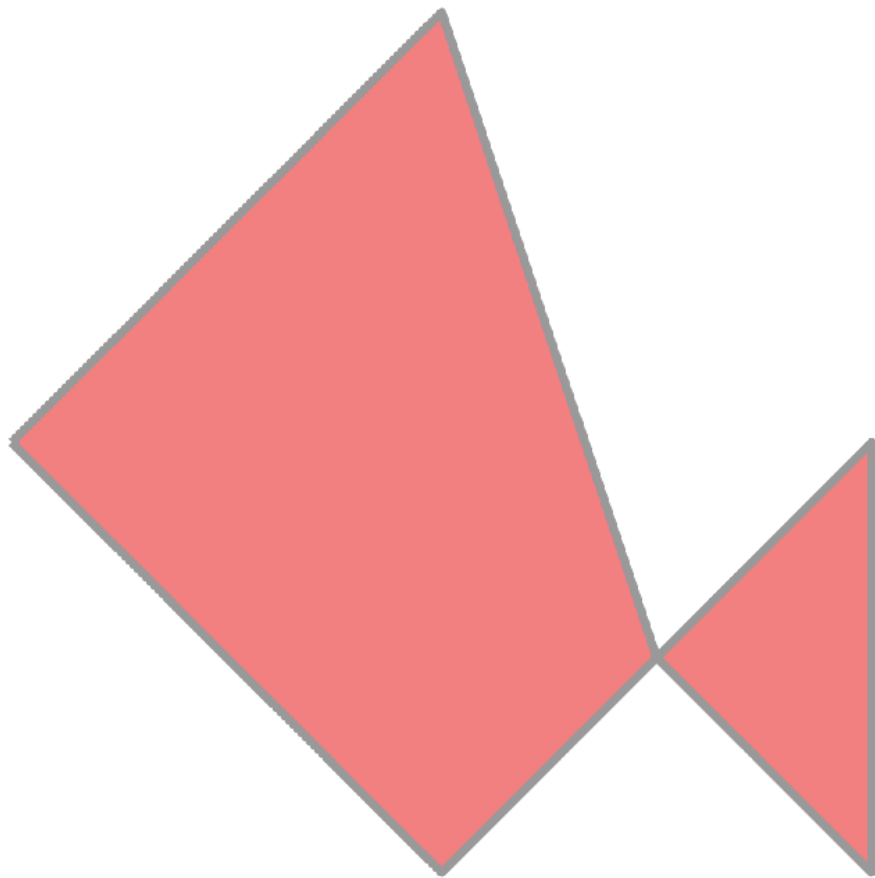
In

```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon
3
4 # Create a polygon from a series of points
5 polygon = Polygon([(0, 0), (1, 1), (1, 0), (0.5, 0.5), (0, 2)])
6 print(f"The polygon: {polygon}", "\n")
7 print(f"Area of the polygon: {polygon.area}", "\n")
8
9 polygon
```

The polygon: POLYGON ((-1 1, 0 0, 1 1, 1 0, 0.5 0.5, 0 2, -1 1))

Area of the polygon: 1.25

Out



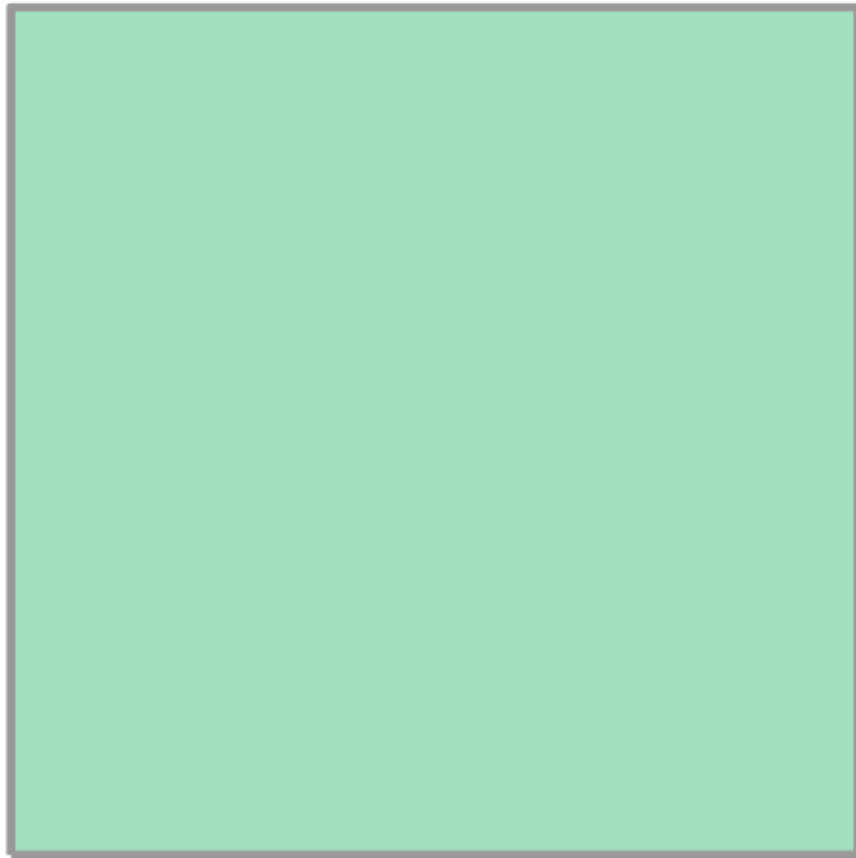
In

```
1 # Compute the envelope (bounding box) of the polygon
2 envelope =
3 print(f'Envelope of the polygon: {envelope}', "\n")
4 print(f'Area of the envelope: "\n")
5
6 envelope
```

Envelope of the polygon: POLYGON ((-1 0, 1 0, 1 2, -1 2, -1 0))

Area of the envelope: 4.0

Out



Convex Hull

The `convex_hull` of a geometry is the smallest convex polygon that can completely contain the geometry. It can be visualized as the shape formed by stretching a rubber band around the outermost points of the geometry. The code below shows how to obtain the convex hull of a polygon, which, in turn, is a convex polygon that follows the outline of the original one.

In

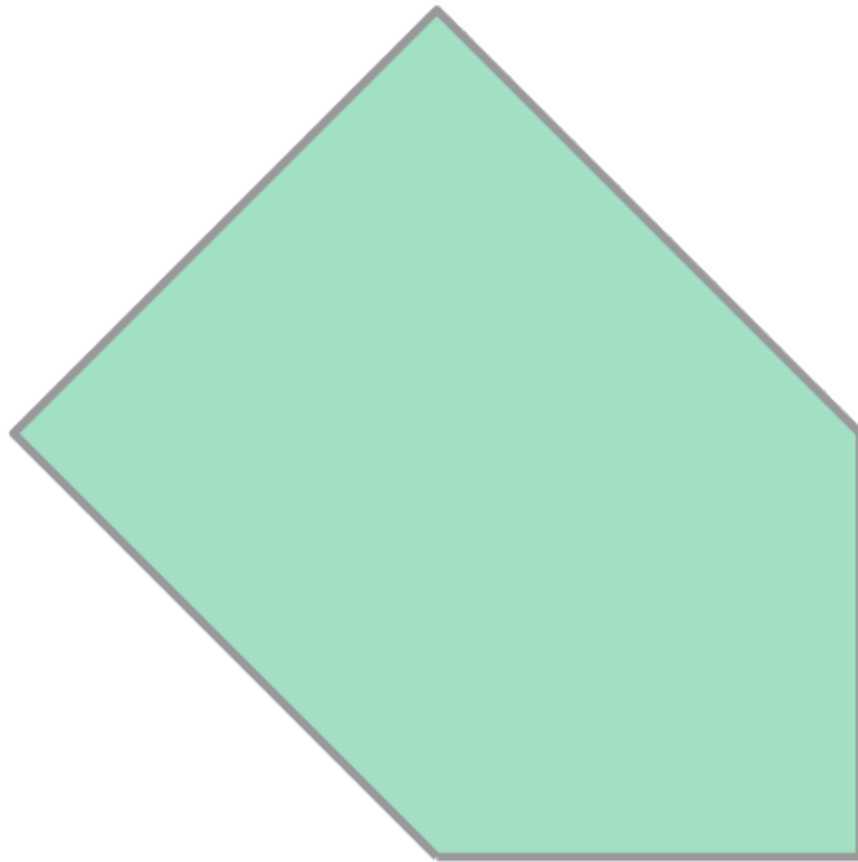
```
1 # Compute the convex hull of the polygon
2 convex_hull =
3 print(f'Convex hull of the polygon: {convex_hull}', "\n")
4 print(f'Area of the convex hull: "\n")

5
6 convex_hull
```

Convex hull of the polygon: POLYGON ((0 0, -1 1, 0 2, 1 1, 1 0, 0 0))

Area of the convex hull: 2.5

Out



Above, we also printed the area of each geometry to illustrate how the envelope and the convex hull alter the original shape. As expected, the total area of the original polygon is the smallest, followed by the convex hull, while the envelope is the largest one.

9. Creating a Bounding Box

A bounding box is a rectangular box that completely contains a geometry or a set of geometries. At first, it looks very similar to the concept of an envelope. However, a bounding box is defined by its minimum and maximum x and y coordinates instead of a seed geometry object. Bounding boxes are useful for spatial indexing, quick overlap checks, and defining the extent of a spatial dataset. In geospatial data science, bounding boxes are often used to quickly determine if objects are likely to intersect or to limit the scope of spatial queries.

Shapely provides a convenient way to create bounding boxes using the box constructor. In this example, we create a rectangular bounding box with a lower left corner at the (0, 0) point and an upper right point at the (1, 2) point.

In

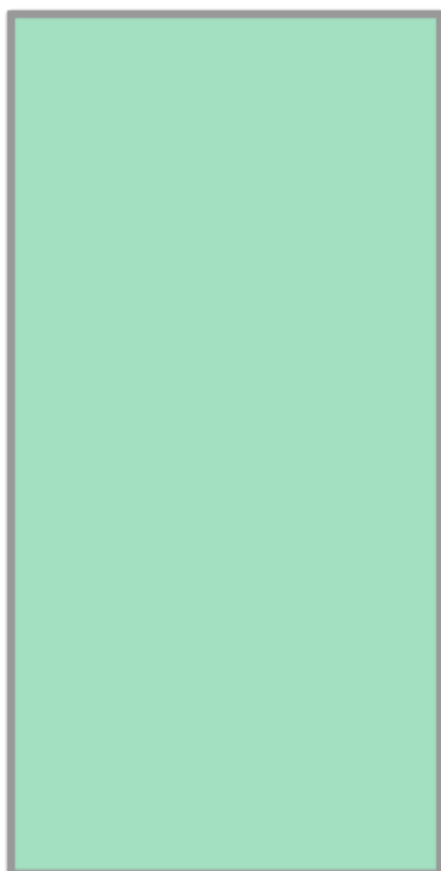
```
1 # Import the box constructor from Shapely
2 from shapely.geometry import box
3
4 # Create a bounding box with specified min and max coordinates
5 bbox =
6
7 # Print the type of the bounding box to confirm its creation
8 print(f'Type of bbox: {type(bbox)}', "\n")
9
10 # Print and show the bounding box
11 print(bbox, "\n")
```

12 bbox

Type of bbox: 'shapely.geometry.polygon.Polygon'>

POLYGON ((1 0, 1 2, 0 2, 0 0, 1 0))

Out



10. Within-test

The within-test is a spatial operation that determines if a geometry is completely inside another geometry. This operation is essential for spatial analysis tasks such as containment checks, identifying if a point lies within a specific region, or determining spatial relationships between different geometries.

Shapely provides the within method to perform this test easily, which we now use to test whether a point at the origin is within a circle that was drawn around the origin with a radius of one and to test if it's within a circle drawn around the (3, 3) point with a unit radius.

In

```
1 # Import the Point constructor from Shapely
2 from shapely.geometry import Point
3
4 # Create a point at coordinates (0, 0)
5 point = Point(0, 0)
6
7 # Create two polygons by buffering points (creating circles)
8 # with a radius of 1 unit
9 polygon_1 = Point(0,
10 polygon_2 = Point(3,
11
12 # Check if the point is within polygon_1 and print the result
13 is_within_polygon_1 =
14 print(f'Point is within polygon_1: {is_within_polygon_1}', "\n")
```

15

16 # Check if the point is within polygon_2 and print the result

17 is_within_polygon_2 =

18 print(f"Point is within polygon_2: {is_within_polygon_2}", "\n")

19

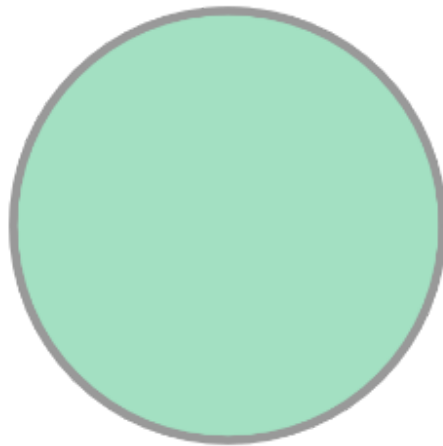
20 # Show the original point and the second polygon together

21

Point is within polygon_1: True

Point is within polygon_2: False

Out



As the first polygon was a circle centered around the original point, it contains that point. However, as the second polygon is a circle with a radius of one and an origin at $(3, 3)$, it does not contain the original point. We illustrate this by showing the joint geometry of the original point and the second circle, which shows how far they are from each other.

11. Distance Calculation

Distance calculation is a fundamental operation in geospatial analysis, allowing us to measure the separation between two geometric objects. This operation can be used to determine the proximity of features, analyze spatial relationships, and perform various spatial queries.

Shapely provides the distance method to compute the distances between different types of geometries, such as points, lines, and polygons. In that order, first, we compute the distance between two points, then the distance between a point and a polygon, and then the distance between two polygons drawn around the initial points. We visualize these geometries by first combining them using the union command and then simply outputting them with their respective code sales.

In

```
1 # Import the Point constructor from Shapely
2 from shapely.geometry import Point
3
4 # Create two points at coordinates (0, 0) and (0, 1)
5 point_1 = Point(0, 0)
6 point_2 = Point(0, 1)
7
8 # Showing the two points on the cell's output
9
```

Out



In

```
1 # Calculate and print the distance between the two points
2 distance_points =
3 print(f"Distance between {point_1} and {point_2}: {distance_points}",
  "\n")
```

Distance between POINT (0 0) and POINT (0 1): 1.0

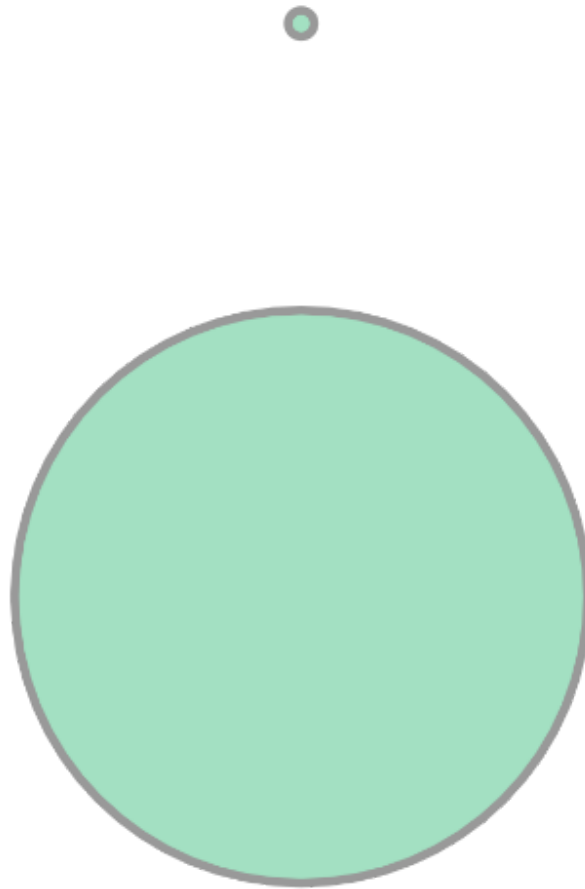
In

```
1 # Create buffers (circles) around the two points with a radius of 0.5 units
2 circle_1 =
3 circle_2 =

4
5 # Calculate and print the distance from the edge of circle_1 to point_2
6 distance_circle_to_point =
7 print(f'Distance from the edge of circle_1 to point_2: \
8     {distance_circle_to_point}', "\n")
9
10 # Showing circle_1 and point_1
11
```

Distance from the edge of circle_1 to point_2: 0.5

Out



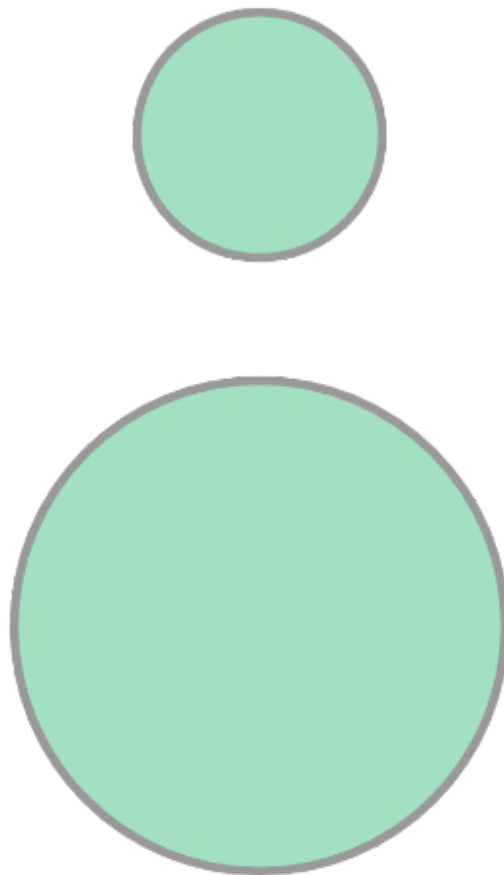
In

```
1 # Calculate and print the distance between circle_1 and circle_2
2 distance_circles =
3 print(f"Distance between the edges of circle_1 and circle_2: \
4     {distance_circles}", "\n")
5
6 # Showing the two circles on the cell's output
```

7

Distance between the edges of circle_1 and circle_2: 0.25

Out



12. Simplifying Geometries

Simplifying a geometry involves reducing the complexity of its shape while retaining its essential form. This process is useful for improving performance during rendering and analysis, especially when working with large or complex shapes. Simplification can help reduce the number of vertices in a geometry, making it easier to process and visualize.

Shapely provides the `simplify` method, which allows us to simplify geometries based on a specified tolerance level. The tolerance defines the maximum distance a point on the simplified geometry can deviate from the original geometry.

In this example, first, we create a polygon with a relatively complicated edge line, and then use two different tolerance parameters of the `simplify` function to see how the simplification goes. While at the tolerance level of 1, we still have a concave object; tolerance level 5 will close the polygon and simplify it down to four edges.

In

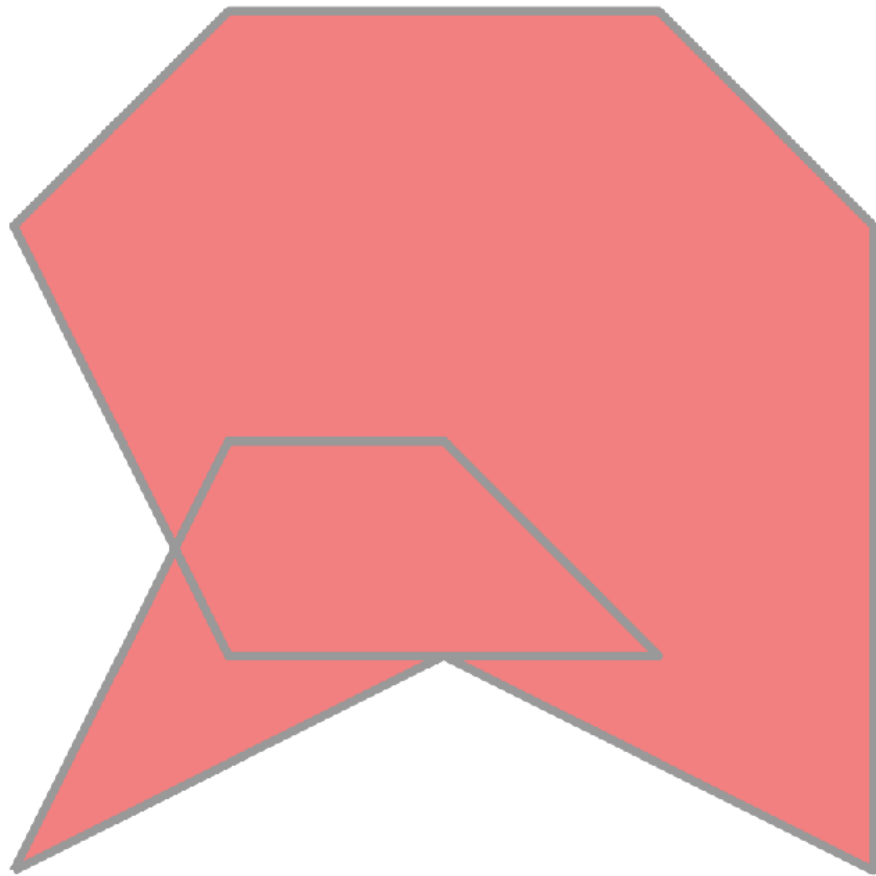
```
1 # Import the Polygon constructor from Shapely
2 from shapely.geometry import Polygon
3
4 # Define the coordinates for the polygon
5 coords = [(0, 0), (2, 1), (4, 0), (4, 3), (3, 4), (1, 4),
6 (0, 3), (1, 1), (3, 1), (2, 2), (1, 2), (0, 0)]
7
8 # Create a polygon from the coordinates
```

```
9 polygon = Polygon(coords)
10
11 # Print and show the original polygon

12 print(f'Original Polygon: {polygon}', "\n")
13 polygon
```

Original Polygon: POLYGON ((0 0, 2 1, 4 0, 4 3, 3 4, 1 4, 0 3, 1 1, 3 1, 2 2, 1 2, 0 0))

Out



In

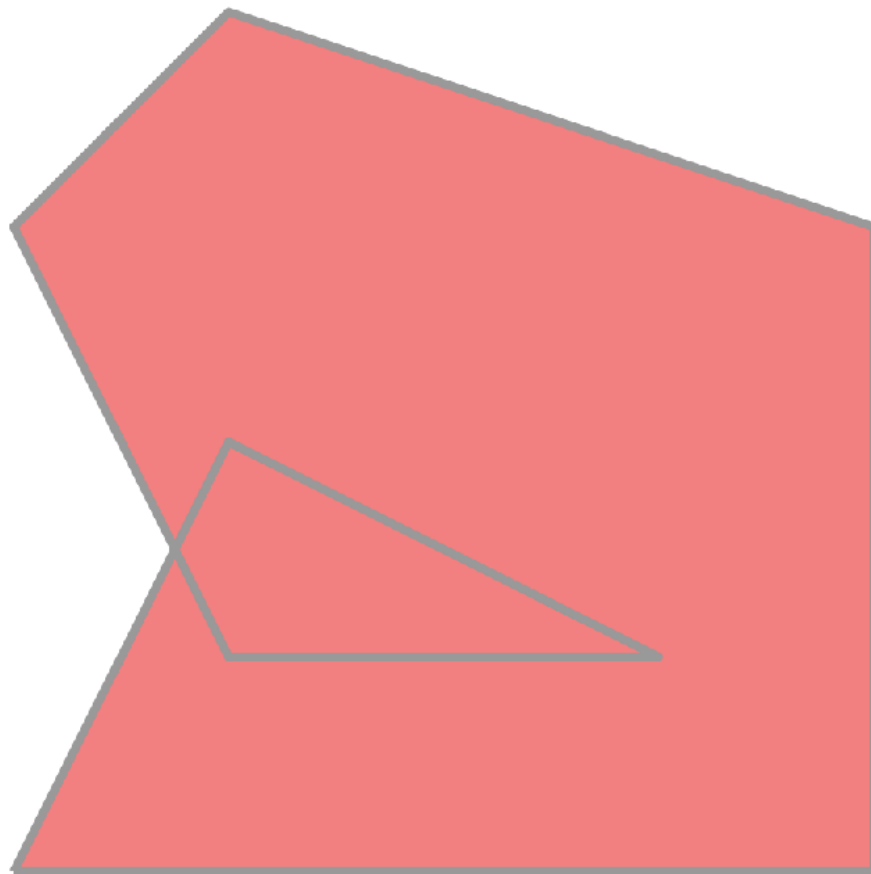
```
1 # Simplify the polygon with a tolerance of 1 unit
2 simplified_polygon_1 =
3
4 # Print and show the simplified polygon

5 print(f'Simplified Polygon with tolerance 1: {simplified_polygon_1}',
  "\n")
```

6 simplified_polygon_1

Simplified Polygon with tolerance 1: POLYGON ((0 0, 4 0, 4 3, 1 4, 0 3, 1 1, 3 1, 1 2, 0 0))

Out



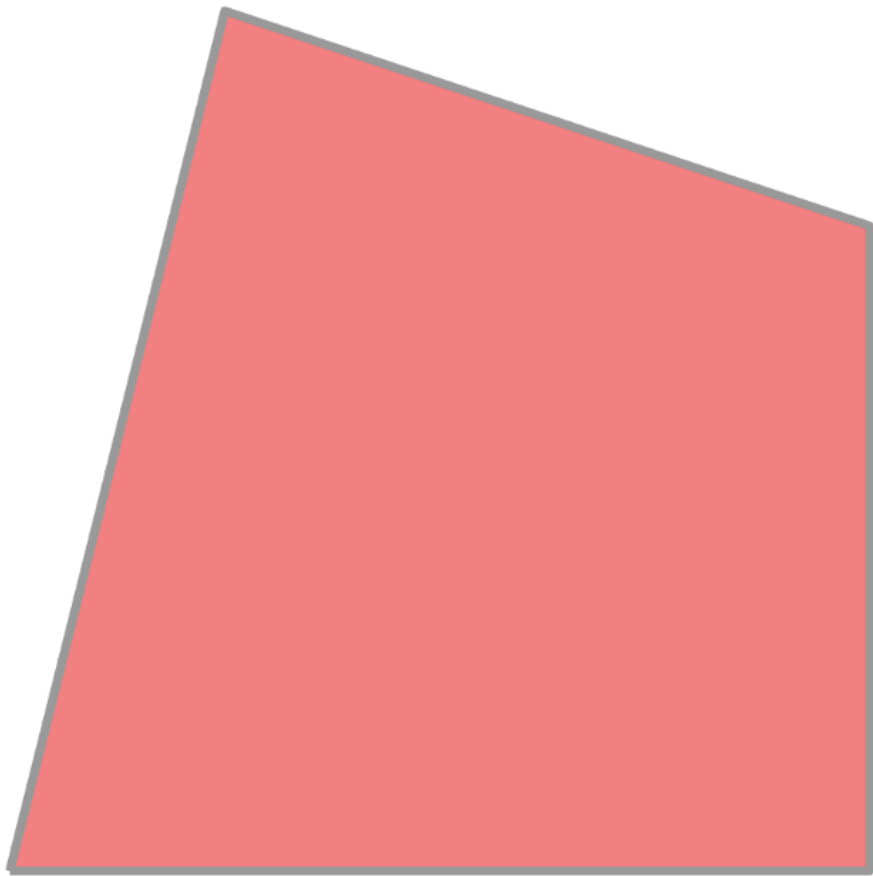
In

```
1 # Simplify the polygon with a tolerance of 5 units
2 simplified_polygon_5 =
3
4 # Print and show the simplified polygon

5 print(f"Simplified Polygon with tolerance 5: {simplified_polygon_5}",
6       "\n")
6 simplified_polygon_5
```

Simplified Polygon with tolerance 5: POLYGON ((0 0, 4 0, 4 3, 1 4, 0 0))

Out



13. 3D Objects in Shapely

Shapely primarily handles 2D geometries, but it also supports 3D geometries by incorporating an additional z-coordinate. These 3D geometries can be used to represent spatial features with elevation or depth information, making them useful for applications in fields such as geography, geology, and environmental science.

We can create and manipulate these geometries by using the usual Shapely tools we saw in the previous examples; as below, we create a 3D point, a 3D line, and a 3D polygon. While the coordinates will define the 3D objects, the current visualization technique only allows for a 2D view. In the following chapters, we will overcome this shortcoming and learn how to visualize geospatial data in 3D.

In

```
1 # Import the necessary constructors from Shapely
2 from shapely.geometry import Point, LineString, Polygon
3
4 # Create a 3D point with coordinates (x, y, z)
5 point_3d = Point(1.0, 2.0, 3.0)
6
7 # Print and show the 3D point
8 print(f'3D Point: {point_3d}', "\n")
9 point_3d
```

3D Point: POINT Z (1 2 3)

Out



In

```
1 # Create a 3D LineString from a list of 3D coordinates
2 line_3d = LineString([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0), (10.0, 15.0, 9.0)])
```

```
3
4 # Print and show the 3D LineString
5 print(f'3D LineString: {line_3d}', "\n")
6 line_3d
```

3D LineString: LINESTRING Z (1 2 3, 4 5 6, 10 15 9)

Out



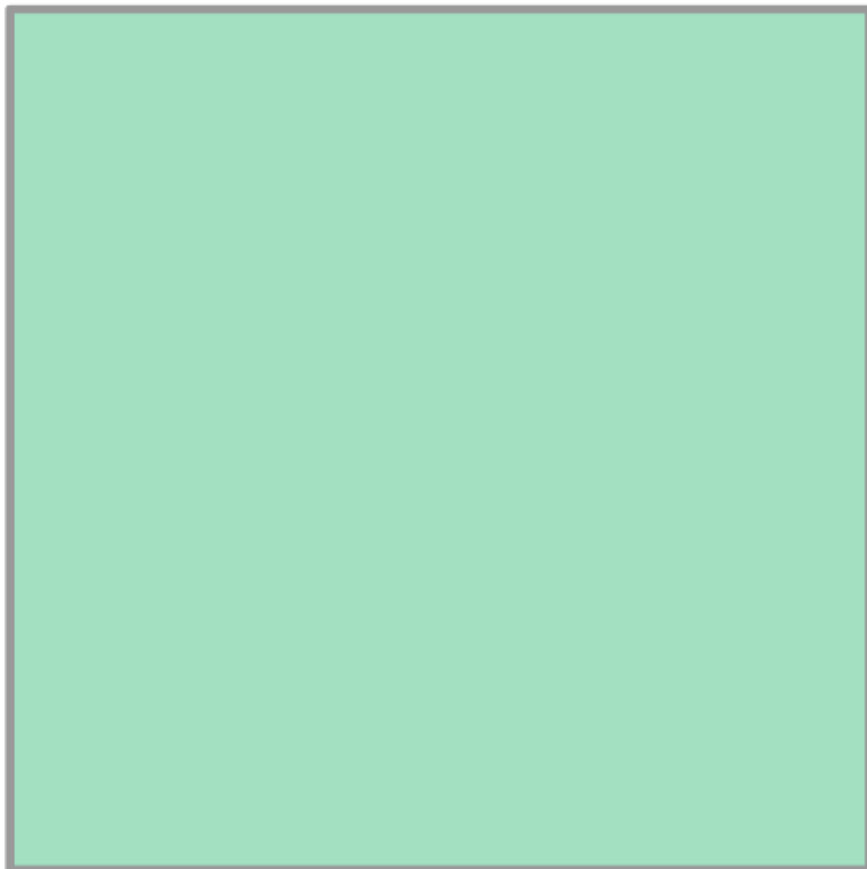
In

```
1 # Create a 3D Polygon from a list of 3D coordinates (the first and last
points must be the same to close the shape)
2 polygon_3d = Polygon([(0.0, 0.0, 0.0), (1.0, 0.0, 1.0),
3 (1.0, 1.0, 2.0), (0.0, 1.0, 3.0),
4 (0.0, 0.0, 0.0)])
5
6 # Print and show the 3D Polygon
```

```
7 print(f'3D Polygon: {polygon_3d}', "\n")
8 polygon_3d
```

3D Polygon: POLYGON Z ((0 0 0, 1 0 1, 1 1 2, 0 1 3, 0 0 0))

Out



Summary on Geometries

In this chapter, we overviewed the core components that make geospatial data and analytics geospatial - geometries. To that end, we explored the Python package Shapely and learned how to use it to create basic geometries, such as points, lines and polygons both in 2D and 3D. We dived into how to transform and manipulate single and multiple geometries as well as how to calculate the centroid point and various enclosing polygons of a geometry. Finally, we learned about geometric functions as, such as computing distance between geometries.

Vector Data in Python



Geospatial data can be broadly categorized into two types: raster and vector data. Raster data represents the world as a grid of equally sized cells, with each cell (or pixel) holding a value representing information such as temperature, elevation, or land cover type. Raster data is typically used for continuous data, such as satellite imagery or digital elevation models (DEMs). In contrast, vector data represents the world using points, lines, and polygons, as we saw with Shapely earlier. Points can represent single locations like bars or cities, lines can represent roads or rivers, and polygons can represent areas such as countries or lakes. Vector data is well-suited for discrete data, where the precise location and shape of geographic features are important.

In this chapter, we will focus on vector data and learn to use the Python library GeoPandas is an extension of the popular data manipulation library Pandas, combined with Shapely's geometry processing capabilities. It allows us to leverage the rich tools for manipulating and analyzing tabular data with

Pandas, along with the versatile geometry operations of Shapely, to provide a robust toolkit for spatial analysis.

14. Querying the Built-in Datasets in GeoPandas

GeoPandas comes with a collection of built-in datasets that we can use for learning and experimentation. While the exact selection of datasets may vary across versions, in general, these datasets are useful for practicing spatial analysis techniques and understanding how to work with GeoPandas' functionalities. We can easily access and query these datasets to explore their content.

Note: while in the GeoPandas version I am using, there are built-in datasets - in the latest release, they removed them. Keep this in mind. If you follow the initial set-up instructions, though, you should be

Let's look at how to query the available built-in datasets in GeoPandas.

In

```
1 # Import the GeoPandas library
2 import geopandas as gpd
3
4 # Query the available built-in datasets in GeoPandas
5 available_datasets =
6
7 # Print the list of available datasets
8 print("Available built-in datasets in GeoPandas:", "\n")
9 print(available_datasets, "\n")
```

Available built-in datasets in GeoPandas:

```
['naturalearth_cities', 'naturalearth_lowres', 'nybb']
```

The previous cell shows how to list the built-in datasets of our GeoPandas version. These datasets are included to help users get started with geospatial analysis without needing to find and load external data. The built-in datasets of this version are called and

This dataset contains point geometries representing major cities around the world. It is sourced from [Natural](#) a public domain map dataset.

This dataset contains polygon geometries representing country boundaries at a relatively low spatial resolution. It is sourced from Natural Earth as well.

This dataset contains polygon geometries representing the borough boundaries of New York City. It stands for "New York Borough Boundaries".

15. Parsing a Data File into a GeoDataFrame

GeoPandas makes it easy to read various geospatial data formats and convert them into which are the core data structures we will be using for spatial analysis throughout this book. We can easily parse data files, such as and other formats, into GeoDataFrames to perform various geospatial operations.

In this example, we will parse the built-in dataset `naturalearth_lowres` from GeoPandas into a GeoDataFrame and display it as a code cell output first by obtaining the location of the file using the `datasets.get_path` function and then use this path as an input for the `read_file` function of GeoPandas. Once parsed, we display the content of the GeoDataFrame.

In

```
1 # Import the GeoPandas library
2 import geopandas as gpd
3
4 # Query the location of the data file
5 file_name = 'naturalearth_lowres'
6 file_path =
7
8 # Print the file path of the sample data
9 print(file_path, "\n")
10
11 # Read a built-in dataset (naturalearth_lowres) into a GeoDataFrame
12 gdf =
```

13

14 # Display the first 3 rows of the GeoDataFrame

15

```
/opt/anaconda3/envs/ox/lib/python3.8/site-  
packages/geopandas/datasets/naturalearth_lowres/naturalearth_lowres.shp
```

Out



In

1 # Display the last 3 rows of the GeoDataFrame

2

Out



As the previous cell outputs show, this table has a component that is easily convertible into a regular Pandas DataFrame. This part includes the name

and code of the countries, the continent they are located in, and is enhanced by the estimated population and GDP levels of each country. However, we see another column there - This geometry column makes the DataFrame a GeoDataFrame, as it contains the location of each data record in a geometric data structure defined by Shapely, using the Polygon and MultiPolygon classes.

16. Accessing the Geometry Column as a GeoSeries

In GeoPandas, each GeoDataFrame contains a special column for geometric data. This column stores the geometries and can be accessed as a GeoSeries is a special type of Pandas Series designed to handle geometric objects, enabling spatial operations and analysis. Here, we take a closer look at that particular column by directly accessing and printing its values of the built-in global map.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame (using the built-in 'naturalearth_lowres'
  dataset for demonstration)
5 gdf =
6
7 # Access the geometry column of the GeoDataFrame
8 geometry =
9 geometry
```

Out

```
0  MULTIPOLYGON (((180.00000 -16.06713, 180.00000...
1  POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...
```



```

2    POLYGON ((-8.66559 27.65643, -8.66512 27.58948...
3    MULTIPOLYGON (((-122.84000 49.00000, -122.9742...
4    MULTIPOLYGON (((-122.84000 49.00000, -120.0000...
...
172   POLYGON ((18.82982 45.90887, 18.82984 45.90888...

173   POLYGON ((20.07070 42.58863, 19.80161 42.50009...
174   POLYGON ((20.59025 41.85541, 20.52295 42.21787...
175   POLYGON ((-61.68000 10.76000, -61.10500 10.890...
176   POLYGON ((30.83385 3.50917, 29.95350 4.17370, ...
Name: geometry, Length: 177, dtype: geometry

```

In

```

1 # Print the type of the geometry column to confirm it is a GeoSeries
2 print(f'Type of the geometry column: {type(geometry)}', "\n")
3
4 # Print the number of elements in the geometry column
5 print(f'Type of the geometry column: {len(geometry)}', "\n")

```

Type of the geometry column: 'geopandas.geoseries.GeoSeries'>

Type of the geometry column: 177

Here, we learned how to quickly access the geometry column in a GeoDataFrame. By accessing the geometry column as a GeoSeries, we can perform various spatial operations and analyses directly on the geometric data within a GeoDataFrame.

17. Creating a GeoDataFrame from Scratch

GeoPandas also allows us to create GeoDataFrames from scratch, which can be particularly useful when we have custom data or want to construct a geospatial dataset manually. A GeoDataFrame is similar to a Pandas DataFrame, but as we saw earlier, it includes a column for geometric data, enabling spatial operations and analyses.

In this example, we'll create a GeoDataFrame from a list of dictionaries containing city names and their locations as Point geometries built from the cities' corresponding geographic coordinates using Shapely.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Define a list of dictionaries with city
6 # names and their geographic locations
7 cities_points = [
8 {'name': 'Budapest', 'geometry': Point(19.0402, 47.4979)},
9 {'name': 'Vienna', 'geometry': Point(16.3738, 48.2082)},
10 {'name': 'Barcelona', 'geometry': Point(2.1734, 41.3851)},
11 {'name': 'New York', 'geometry': Point(40.7128, 74.0060)},
12 {'name': 'Los Angeles', 'geometry': Point(34.0522, -118.2437)},
13 {'name': 'Helsinki', 'geometry': Point(24.9354, 60.1695)},
14 {'name': 'Dublin', 'geometry': Point(53.3498, -6.2603)}
```

```
15 {'name': 'London', 'geometry': 51.5074}},  
16 {'name': 'Amsterdam', 'geometry': Point(4.9041, 52.3676)}  
  
17 ]  
18  
19 # Create a GeoDataFrame from the list of dictionaries  
20 gdf_cities =  
21  
22 # Show the GeoDataFrame  
23 gdf_cities
```

Out



This cell output shows the example GeoDataFrame we just created, containing the point-level location and name of these selected cities.

18. Creating a GeoDataFrame from a DataFrame

After creating a GeoDataFrame from scratch, let's see how we can convert Pandas to GeoPandas. It's important to know that GeoPandas allows us to easily convert a Pandas DataFrame containing geographic coordinates into a GeoDataFrame. This process involves creating geometric objects (such as points) from coordinate columns using Shapely and then constructing a GeoDataFrame that includes these geometries. This is useful for transforming traditional tabular data into a geospatial format for spatial analysis.

In

```
1 # Import the necessary libraries
2 import pandas as pd
3 import geopandas as gpd
4 from shapely.geometry import Point
5
6 # Define a list of dictionaries with city
7 # names and their geographic locations
8 df_cities =
9 {'name': 'Budapest', 'longitude': 19.0402, 'latitude': 47.4979},
10 {'name': 'Vienna', 'longitude': 16.3738, 'latitude': 48.2082},
11 {'name': 'Barcelona', 'longitude': 2.1734, 'latitude': 41.3851},
12 {'name': 'New York', 'longitude': 'latitude': 40.7128},
13 {'name': 'Los Angeles', 'longitude': 'latitude': 34.0522},
14 {'name': 'Helsinki', 'longitude': 24.9354, 'latitude': 60.1695},
```

```
15 {'name': 'Dublin', 'longitude': 'latitude': 53.3498},
16 {'name': 'London', 'longitude': 'latitude': 51.5074},
17 {'name': 'Amsterdam', 'longitude': 4.9041, 'latitude': 52.3676}
18 ])
19
20 # Display the first 5 rows of the DataFrame
21 print("Pandas DataFrame:", "\n")
22
```

Pandas DataFrame:

| | | | | | |
|------------|------------|------------|------------|------------|------------|
| DataFrame: | DataFrame: | DataFrame: | DataFrame: | DataFrame: | DataFrame: |
| | | | | | |

In

```
1 # Create a list of Shapely Point objects from the long and lat columns
2 geometry = [Point(xy) for xy in \
3 zip(df_cities['longitude'], df_cities['latitude'])]
4
5
6 # Print the first five geometries we created
7 geometry[0:5]
```

Out

```
[(19.04 47.498)>,  
(16.374 48.208)>,  
(2.173 41.385)>,  
(-74.006 40.713)>,  
(-118.244 34.052)>]
```

In

```
1 # Create a GeoDataFrame from the DataFrame, including the geometry  
column  
2 gdf_cities =  
3  
4 # Optionally, drop the original longitude and latitude columns  
5 gdf_cities = gdf_cities.drop(['longitude', 'latitude'], axis=1)  
6  
7 # Display the first 5 rows of the GeoDataFrame  
8 print("\nGeoPandas GeoDataFrame:")  
9
```

GeoPandas GeoDataFrame:

Out



In this section, we learned how to convert a Pandas DataFrame containing city names and their geographic coordinates as float numbers into a GeoPandas GeoDataFrame by creating geometric Point objects from the longitude and latitude columns via Shapely.

19. Writing a GeoDataFrame to a File

GeoPandas allows us to easily export GeoDataFrames to various GIS file formats, making it simple to share, store, and visualize our geospatial data. The most common formats include [Shapefile](#) and among others. Exporting GeoDataFrames to these formats ensures compatibility with many GIS tools and applications.

In the following example, we will recreate the global cities' GeoDataFrame using the previously defined cities_points dictionary and export it into these two different geospatial file types using the to_file method.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Create the GeoDataFrame
6 gdf_cities =
7
8 # Write the GeoDataFrame to a Shapefile
9
10
11 # Write the GeoDataFrame to a GeoJSON file
12
13
```


14 # Confirmation messages

15 print("GeoDataFrame has been written to 'Cities.shp'.", "\n")

16 print("GeoDataFrame has been written to 'Cities.geojson'.", "\n")

GeoDataFrame has been written to 'Cities.shp'.

GeoDataFrame has been written to 'Cities.geojson'.

20. Bounds of a GeoSeries

The bounds of a GeoSeries, storing the geometric information of a GeoDataFrame, provide the minimum bounding box that completely contains each geometry in the series. This bounding box is defined by the minimum and maximum x and y coordinates (minx, miny, maxx, maxy) for each geometry. Knowing the bounds of geometries can be useful for various spatial analyses, such as determining the extent of spatial features and performing spatial indexing. Let's see how to obtain using the sample global map data.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Read a built-in dataset (naturalearth_lowres) into a GeoDataFrame
6 gdf =
7
8 # Display the first five countries in the GeoDataFrame
9
10
11 # Access the bounds of the geometry column
12 bounds =
13
14 # Display the bounds of the first five geometries
15 print("Bounds of the geometries in the GeoDataFrame:", "\n")
```



Bounds of the geometries in the GeoDataFrame:

Out



As this example quickly illustrates, by accessing the bounds of a GeoSeries, we can determine the spatial extents of individual geometries, which is useful for spatial indexing, visualization, and various analytical tasks in geospatial analysis.

21. Area and Perimeter Computation

As GeoPandas is building on Shapely, it has the ability to calculate the area and perimeter length of geometries stored in a GeoDataFrame. Area calculation is especially useful for polygons, which represent the surface area of geographic features. The length and the area are computed in the units of the coordinate reference system (CRS) used by the geometries. Later on, we will dedicate an entire chapter to CRS, so for now, just leave it as it is.

In the following examples, we will import the built-in global country map data file, store it in a GeoDataFrame, and then call its built-in length and area computation commands.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame (using the built-in
5 # 'naturalearth_lowres' dataset for demonstration)
6 gdf =
7
8 # Display the first 3 rows of the GeoDataFrame
9 print("First 3 rows of the GeoDataFrame:", "\n")
10
```

First 3 rows of the GeoDataFrame:

| | | |
|---------------|---------------|---------------|
| GeoDataFrame: | | |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

In

```
1 # Calculate the length of each geometry in the GeoDataFrame
2 length =
3
4 # Print the areas of the geometries
5 print("Data type of the length:", type(length), "\n")
6
7 # Print the areas of the geometries
8 print("Length of the geometries as country perimeters:", "\n")
9 length
```

Data type of the length: 'pandas.core.series.Series'>

Length of the geometries as country perimeters:

Out

```
0    8.991010
1   37.260671
2   27.662143
3  916.062855
4  356.977119
...
172 16.326071
173  5.679511
174  4.938975
175  3.387105
176 37.450164
Length: 177, dtype: float64
```

In

```
1 # Calculate the area of each geometry in the GeoDataFrame
2 areas =
3
4 # Print the areas of the geometries
5 print("Data type of the areas:", type(areas), "\n")
6
7 # Print the areas of the geometries
8 print("Areas of the geometries as country areas:", "\n")
9 areas
```

Data type of the areas: 'pandas.core.series.Series'>

Areas of the geometries as country areas:

Out

```
0    1.639511
1    76.301964
2     8.603984
3   1712.995228
4   1122.281921
...
172   8.604719
173   1.479321
174   1.231641
175   0.639000
176   51.196106
```

Length: 177, dtype: float64

By calculating the area of geometries in a GeoDataFrame, we can perform various spatial analyses that require surface area measurements, such as land use analysis, habitat estimation, and resource management.

22. Simple Visualization with GeoPandas

GeoPandas makes it easy to visualize geospatial data by providing built-in plotting capabilities. Using the `plot` method, we can quickly create simple maps to visualize the geometries in a `GeoDataFrame`. This functionality is helpful for creating quick visual representations of spatial data.

In this example, first, we read the global country map data and then quickly visualize the `GeoDataFrame`.

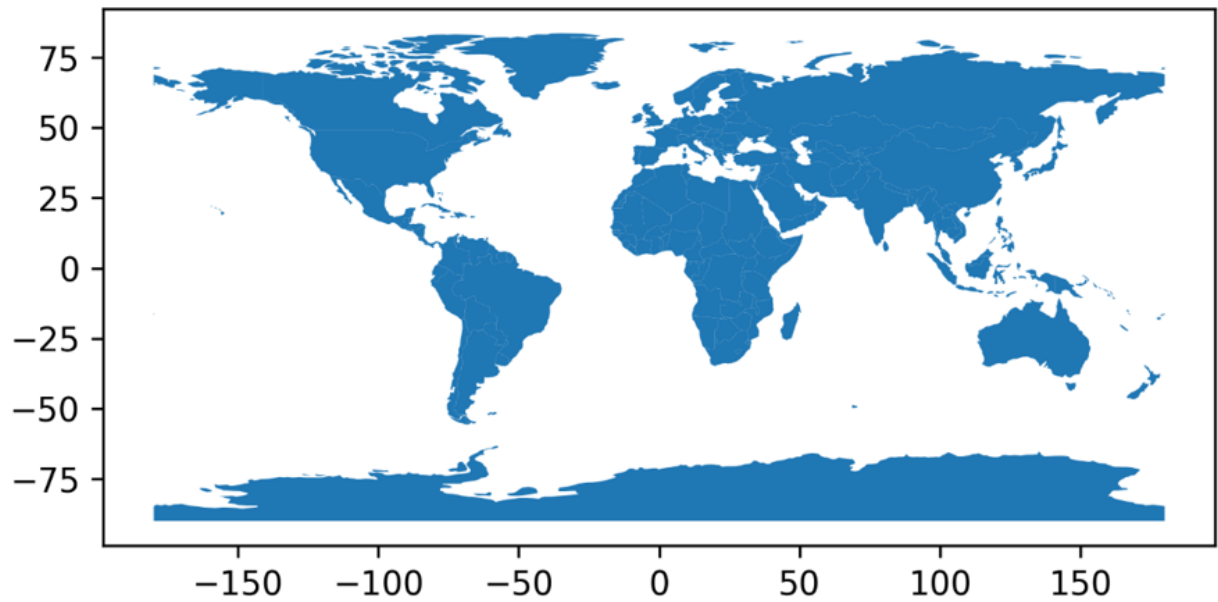
While we are already using the visualization library Matplotlib behind the scenes, we will dive deeper into geospatial data visualization in the upcoming chapters.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame (using the built-in
5 # 'naturalearth_lowres' dataset for demonstration)
6 gdf =
7
8 # Plot the GeoDataFrame
9
```

Out

>



By using GeoPandas' built-in plotting capabilities, we can quickly create visualizations of our geospatial data. In the later dedicated chapter, we will further explore how these maps can be customized for advanced spatial analytics.

23. Buffering a GeoDataFrame

Buffering is a common spatial operation implemented in Shapely and, consequently, inherited by GeoPandas. Geometry buffering involves creating zones around geometric features. In GeoPandas, we can easily buffer geometries in a GeoDataFrame using the `buffer` method, which is useful for various spatial analyses such as proximity analysis, impact assessments, and creating buffer zones around specific spatial features.

In the following code block, we first create a sample GeoDataFrame consisting of the locations of a few selected global cities using the previously defined `cities_points` dictionary. Then, we create a copy of the GeoDataFrame, which we will modify by applying a buffer zone with a radius of 5 units. As a comparison, we visualize both the unbuffered and buffered GeoDataFrames.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Create the GeoDataFrame
6 gdf_cities =
7
8 # Plot the original GeoDataFrame
9
10
11 # Create a copy of the GeoDataFrame for buffering
12 gdf_cities_buffered =
13
14 # Apply a buffer of 5 units around each geometry
15 gdf_cities_buffered['geometry'] =
16
17 # Display the buffered GeoDataFrame
18 print("Buffered GeoDataFrame:", "\n")
19 display(gdf_cities_buffered)
20
```

21 # Plot the buffered GeoDataFrame

22

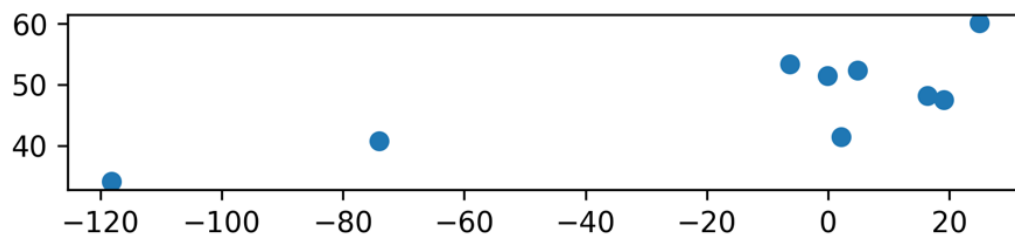
Buffered GeoDataFrame:

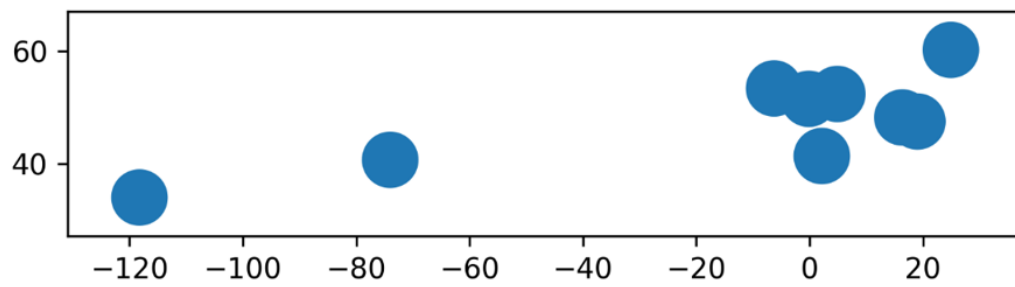
| | | | | | |
|---------------|---------------|---------------|---------------|---------------|---------------|
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

| | | | |
|---------------|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

Out

>





By buffering a GeoDataFrame relying on the underlying Shapely library, we can create zones around our spatial features, which is useful for various analyses such as determining areas of influence, impact zones, and proximity analysis. As this example shows, with buffering, we can replace all points stored in a GeoDataFrame with circles of a given radius.

24. Spatial Join with GeoPandas

Spatial join is a powerful operation that combines two GeoDataFrames based on their spatial relationship. This operation is useful for integrating different spatial datasets and performing complex spatial analyses. GeoPandas provides the `sjoin` method to perform spatial joins, allowing us to merge GeoDataFrames based on spatial relations like intersection, within, and contains.

In the next example, we will illustrate spatial join by combining two of the data sets we have been previously using in this section. First, we read the global country-level data again, containing the polygon boundaries of each country. Second, we re-create the global city selection table containing the point locations of each city using the `cities_points` dictionary.

In

```
1 # Import the necessary library
2 import geopandas as gpdd
3 from shapely.geometry import Point
4
5 # Load a sample GeoDataFrame (using the built-in
6 # 'naturalearth_lowres' dataset for countries)
7 gdf_countries =
8
9 # Create the GeoDataFrame for cities
10 gdf_cities =
```

Then, we apply several data preparation steps, such as selecting and renaming the target columns within these DataFrames.

In

```
1 # Filter and prepare the countries GeoDataFrame
2 gdf_countries_filtered = gdf_countries[['name', \
3
4
5 # Filter and prepare the cities GeoDataFrame
6 gdf_cities_filtered = gdf_cities[['name', \
7
8
9 # Display the first 3 rows of the filtered countries GeoDataFrame
10 print("Filtered countries GeoDataFrame:", "\n")
11
12
13 # Display the first 3 rows of the filtered cities GeoDataFrame
14 print("Filtered cities GeoDataFrame:", "\n")
15
16
17 # Print the number of entries in the filtered GeoDataFrames
18 print(f'Number of countries: {len(gdf_countries_filtered)}', "\n")
19 print(f'Number of cities: {len(gdf_cities_filtered)}', "\n")
```

Filtered countries GeoDataFrame:

| | | | |
|---------------|---------------|---------------|---------------|
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

Filtered cities GeoDataFrame:

| | | | |
|---------------|---------------|---------------|---------------|
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

Number of countries: 177

Number of cities: 9

Now, we perform the spatial joins. First, we join the cities to the countries, and then, we join the countries to the cities. The order will define the type of output we get from the spatial joins.

In

```

1 # Perform a spatial join: cities within countries

2 cities_within_countries =
3     gdf_countries_filtered,
4
5
6
7 print("Number of cities matched to countries:", \
8     "\n")
9

```

```
10 print("Cities within countries:", "\n")
11 display(cities_within_countries)
12
```

Number of cities matched to countries: 9

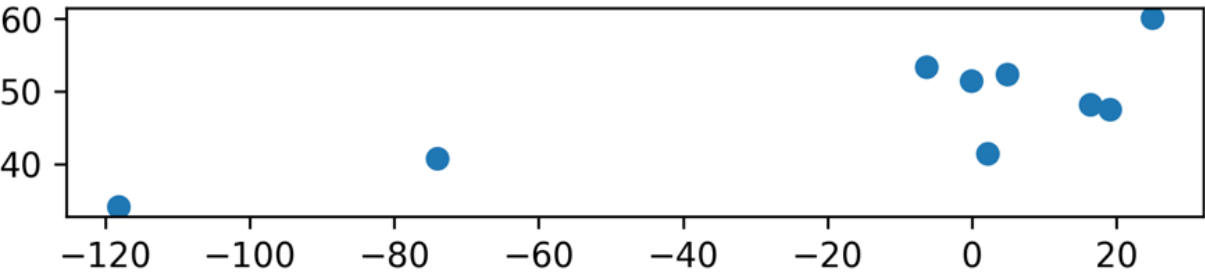
Cities within countries:

| | | | | |
|------------|------------|------------|------------|----------------------------------|
| countries: | countries: | countries: | countries: | countries: countries: countries: |
| | | | | countries: |

| | | | | |
|-----------------------|------------|------------|------------|------------|
| countries: countries: | countries: | countries: | countries: | countries: |
| countries: countries: | | | countries: | |

Out

>



In

```
1 # Perform a spatial join: countries containing cities
```



```

2 countries_containing_cities =
3     gdf_cities_filtered,
4
5
6
7 print("Number of countries containing cities:", \
8     "\n")

9
10 print("Countries containing cities:", "\n")
11 display(countries_containing_cities)
12

```

Number of countries containing cities: 8

Countries containing cities:

| | | | | | |
|---------|-----------------|-----------------|---------|---------|---------|
| cities: | cities: cities: | cities: cities: | cities: | cities: | cities: |
|---------|-----------------|-----------------|---------|---------|---------|

| | | | |
|---------|---------|---------|---------|
| cities: | cities: | cities: | cities: |
|---------|---------|---------|---------|

Out

>



On the one hand, these results show that the spatial join successfully matched all nine cities to their respective countries, so the number of cities matched to countries equals the number of cities in the original GeoDataFrame. Since all cities were matched to a country, the `cities_within_countries` map displays all the cities we initially had.

On the other hand, we filtered down the global list of 177 countries to only eight countries by joining the nine cities, as both New York City and Los Angeles belong to the US. As a result, the `countries_containing_cities` map contains only eight countries.

25. Overlaying GeoDataFrames

Overlaying GeoDataFrames involves combining two spatial datasets based on their spatial relationship to create a new GeoDataFrame. GeoPandas provides the `overlay` method to perform these operations, allowing us to integrate and analyze spatial data efficiently.

In the following example, we first prepare the input data, which includes both the built-in world map and our example selection of cities using the `cities_points` dictionary. Then, we buffer the GeoDataFrame containing the cities, as demonstrated in the Buffering a GeoDataFrame section. Finally, we call the `overlay` method and use it to create a GeoDataFrame containing the intersection geometries of the cities' buffer zones and the country polygons they intersect.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Load a sample GeoDataFrame
6 gdf_countries =
7
8 # Create the GeoDataFrame for cities
9 gdf_cities =
10
11 # Buffer the cities GeoDataFrame to create buffer zones around each city
12 gdf_cities_buffered =
13
14 gdf_cities_buffered['geometry'] =
```

```

14
15 # Filter and prepare the countries GeoDataFrame
16 gdf_countries_filtered = gdf_countries[['name',
17
18
19 # Perform an overlay operation to find the intersection of buffered cities
with countries
20 gdf_over =
21         gdf_cities_buffered,
22
23
24 # Display the first 10 rows of the overlay result
25 print("Overlay result (first 10 rows):", "\n")
26
27
28 # Plot the overlay result
29

```

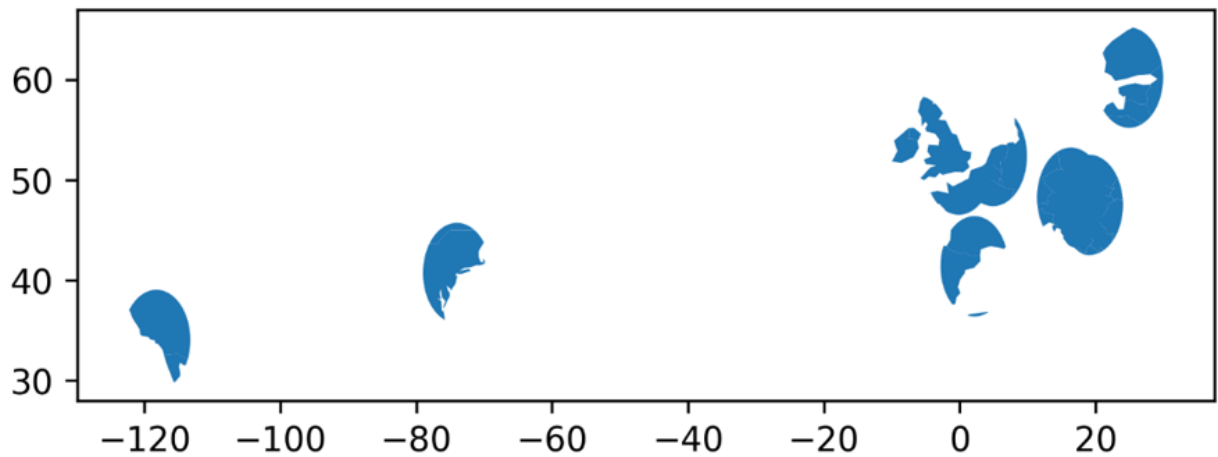
Overlay result (first 10 rows):

| | | | | | | | | |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|

| | | | | | | | | |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |
| rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): | rows): |

Out

>



By overlaying GeoDataFrames, we can create new GeoDataFrames by combining two DataFrames based on intersection areas, enabling more complex and insightful geospatial analyses. As during overlay, we capture the intersecting parts of geometries, the previous code block results in polygons, which are the unions of circles (buffered city-location points) and countries, resulting in these strange, circularly cropped country fragments on the world map.

26. Dissolving Polygons

Dissolving polygons is a spatial operation that combines multiple geometries within a GeoDataFrame based on a specified attribute. This process merges geometries that share the same value for the specified attribute, creating a single, unified geometry for each unique value. Dissolving is helpful for aggregating spatial data and simplifying the representation of geographic features.

To demonstrate how dissolving a GeoDataFrame works, we will merge all countries within the `naturalearth_lowres` dataset that belong to the same continent. According to this dissolve operation, the final output of this cell will be a global map of continents, not countries, as the country boundaries will be dissolved along the shared attributes - joint continents. To better illustrate the effect of the operation, we set the edge color of each polygon to red in the plot method.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame
5 gdf =
6
7 # Plot the country-level world map
8 = 'red')
9
10 # Dissolve polygons by the 'continent' attribute to
11 # combine countries into continents
12
12 gdf_dissolved =
13
14 # Display the dissolved GeoDataFrame
15 print("Dissolved GeoDataFrame:", "\n")
16 display(gdf_dissolved)
17
18 # Print the size of the GeoDataFrame before and after the operation
```

```

19 print("Number of records before the dissolve: ", len(gdf), "\n")
20 print("Number of records before the dissolve: ", len(gdf_dissolved), "\n")
21
22 # Plot the dissolved GeoDataFrame to visualize the combined continents
23 = 'red')

```

Dissolved GeoDataFrame:

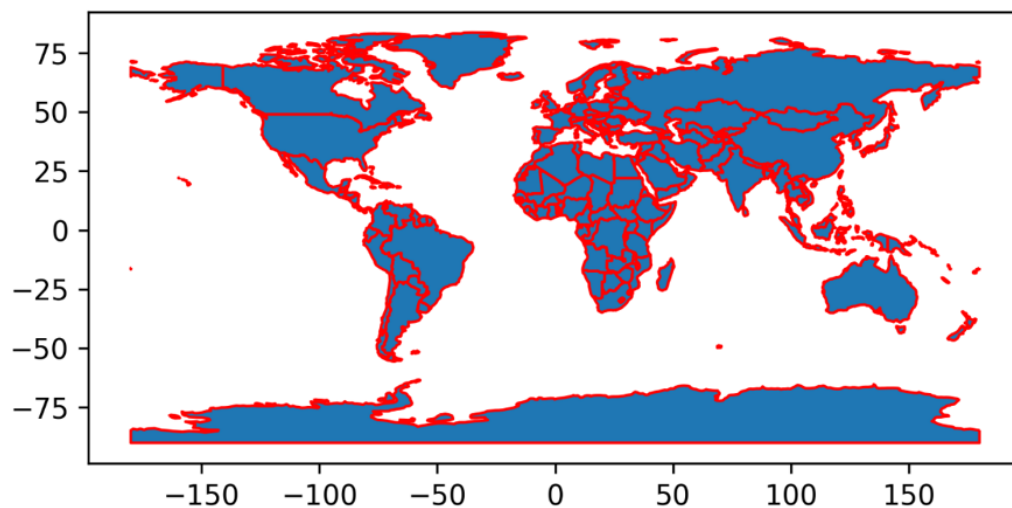
| | | | | | |
|---------------|---------------|---------------|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | | | |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

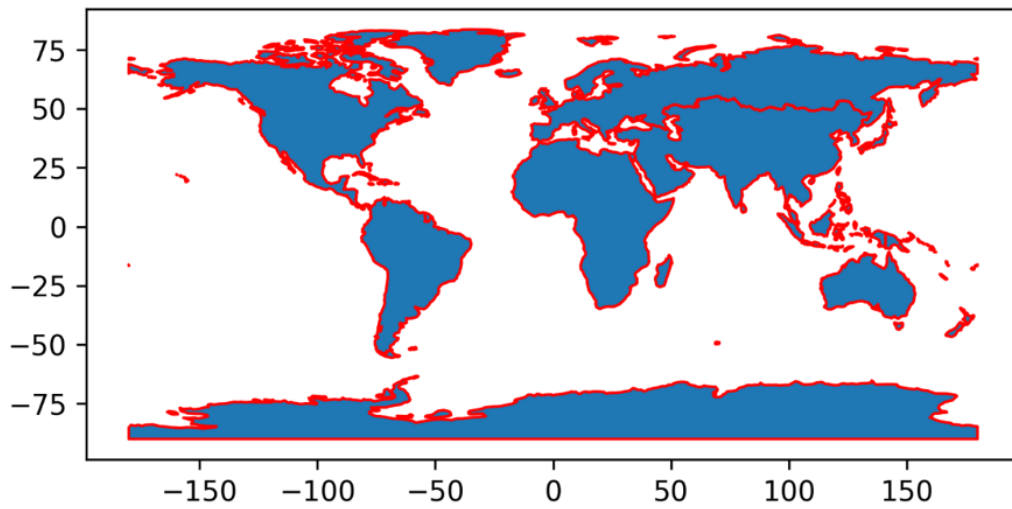
Number of records before the dissolve: 177

Number of records before the dissolve: 8

Out

>





The output images show the country-level original and continent-level dissolved world maps, where every polygon's edge line is marked in red. This edge coloring highlights the countries in the first plot while only showing continent boundaries in the second.

27. Splitting Geometries in a GeoDataFrame

As we can merge geometries, for instance, by dissolving them based on mutual attributes, we can also split them. In particular, splitting geometries in a GeoDataFrame involves breaking multi-part geometries (such as MultiPolygons or MultiLineStrings) into their individual components. This process is useful for detailed analysis and manipulation of each component separately. GeoPandas provides the `explode` method to achieve this, which splits multi-part geometries into single geometries and creates a new row for each part. By splitting geometries in a GeoDataFrame, we can analyze and manipulate each component of multi-part geometries separately, enabling more detailed and granular spatial analyses.

To demonstrate this, we first read the country-level world map dataset. By displaying its first ten rows, we can already see that there are multiple MultiPolygons within. Once we apply the `explode` command, we will see a significant increase in the number of records within the resulting GeoDataFrame, with each geometry stored as a Polygon. This illustrates that all the MultiPolygons were efficiently split into individual Polygons, each polygon inheriting all the other attributes of the initial data record. This transformation allows for more granular spatial analysis and manipulation.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame

5 gdf =
6
7 # Display the original GeoDataFrame
8 print("Original GeoDataFrame:", "\n")
9
10
11
12 # Split multi-part geometries into their individual components
13 # using the explode method
14 gdf_exploded =
15
```

```

16 # Display the exploded GeoDataFrame
17 print("Exploded GeoDataFrame:", "\n")
18
19
20 # Print the size of the GeoDataFrame before and after the operation
21 print("Number of records before the explode: ", len(gdf), "\n")
22 print("Number of records before the explode: ", len(gdf_exploded), "\n")

```

Original GeoDataFrame:

| | | | | | |
|---------------|---------------|---------------|---------------|---------------|---------------|
| GeoDataFrame: | | | | | |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

| | | | |
|---------------|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

Exploded GeoDataFrame:

| | | |
|---------------|-----------------------------|-----------------------------|
| GeoDataFrame: | GeoDataFrame: GeoDataFrame: | GeoDataFrame: GeoDataFrame: |
| | GeoDataFrame: GeoDataFrame: | GeoDataFrame: GeoDataFrame: |
| | GeoDataFrame: | GeoDataFrame: |

| | | | | |
|---------------|---------------|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| | | | | |

| | | | | |
|---------------|---------------|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

| | | |
|---------------|---------------|---------------|
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |
| GeoDataFrame: | GeoDataFrame: | GeoDataFrame: |

Number of records before the explode: 177

Number of records before the explode: 287

28. Applying Simple Functions on GeoDataFrames

Applying functions to the geometries in a GeoDataFrame allows us to compute new attributes based on spatial properties. GeoPandas makes it easy to use Python's list comprehensions and the apply method to derive additional columns, such as geometric properties like length and area. In the following example, we will use these two methods to add new columns to our GeoDataFrame containing the global map. These new columns will contain the length and area, as well as the exact type of geometries.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample GeoDataFrame
5 gdf =
6
7 # Filter the GeoDataFrame to include only
8 # the necessary columns for this example
9 gdf_countries_filtered = gdf[['name',
10
11
12 # Display the GeoDataFrame
13
```

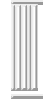
Out



In

```
1 # Calculate the length of each geometry and add it as a new column
2 gdf_countries_filtered['geometry_length'] = \
3
4         lambda x:
5
6 # Calculate the area of each geometry and add it as a new column
7 gdf_countries_filtered['geometry_area'] = \
8
9         lambda x:
10
11 # Calculate the type of each geometry and add it as a new column
12 gdf_countries_filtered['geometry_type'] = \
13
14         lambda x:
15
16 # Display the GeoDataFrame
17
```

Out



By applying simple functions to GeoDataFrames, we can easily compute and add new geometric properties, enhancing our spatial data analysis capabilities.

29. Generating Random Synthetic Data

Generating random synthetic data can be useful for testing, simulation, and demonstrating geospatial techniques. In this example, we will use the world map GeoDataFrame, determine its bounding box, and generate 1000 points uniformly randomly placed within that bounding box using Once the points are generated and stored in a GeoDataFrame, we will visualize the results.

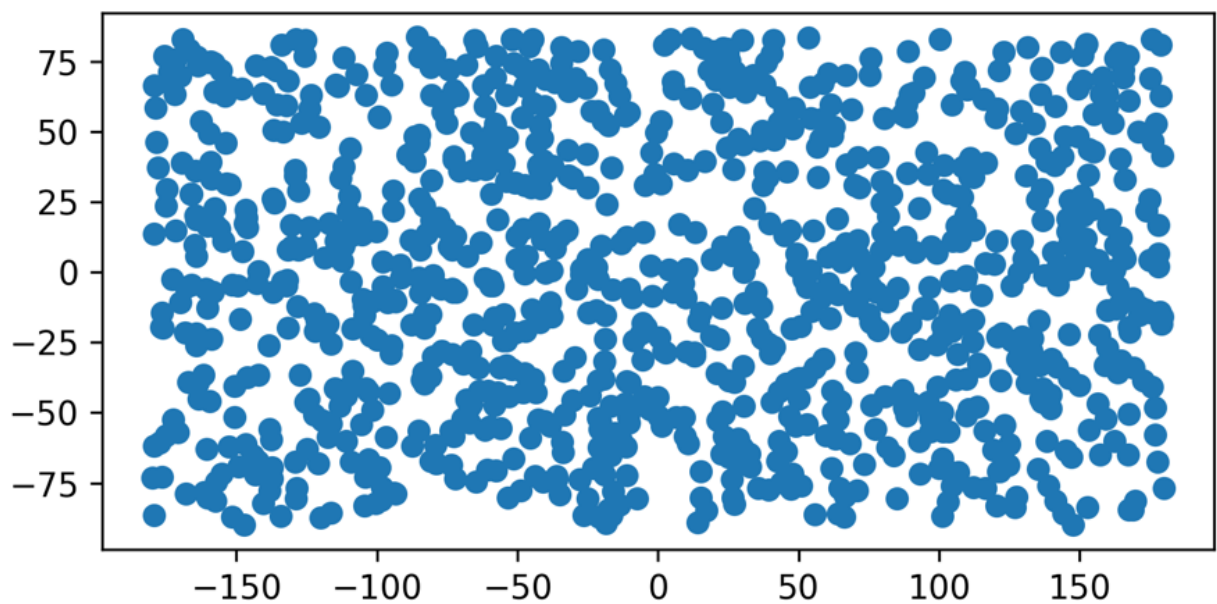
In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 import numpy as np
4 from shapely.geometry import Point
5
6 # Load a sample GeoDataFrame
7 gdf =
8
9 # Calculate the minimum and maximum bounds of the GeoDataFrame
10 bounds =
11 minx, miny, maxx, maxy = bounds
12
13 # Initialize an empty list to store random points
14 random_points = []
15
16 # Generate 1000 random points within the bounds
17 for _ in range(1000):
18     x = maxx)
19     y = maxy)
```

```
20     y))
21
22 # Create a GeoDataFrame from the list of random points
23 gdf_rand =
24
25 # Plot the GeoDataFrame with random points
26
```

Out

>



By generating random synthetic data, we can create test datasets for spatial analysis, simulations, and demonstrations, enhancing our ability to develop and test geospatial techniques.

30. Counting Points in Polygons

Counting points within polygons is a common spatial analysis task that helps determine how many point features fall within each polygon feature. This is useful for various applications such as population density analysis, event clustering, and resource allocation. GeoPandas provides tools to perform spatial joins and group by operations to achieve this.

In this exercise, we will combine several techniques from previous sections. First, we will create 10,000 random points spread across the global map. Then, we will use a spatial join to match each point to one of the countries and count the number of points within each country. This task is similar to the previous spatial join example but uses a data size significantly closer to real-life expectations. Finally, we will merge the prepared join to add the point count value to the original DataFrame to prepare a data table appropriate for further spatial analysis.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point
4 import numpy as np
5
6 # Load a sample GeoDataFrame
7 gdf_countries =
8
9 # Calculate the minimum and maximum bounds of the GeoDataFrame
```

```
10 bounds =
11 minx, miny, maxx, maxy = bounds
12
13 # Initialize an empty list to store random points
14 random_points = []
15
16 # Generate 100000 random points within the bounds
17 for _ in range(10000):
18     x = maxx)
19     y = maxy)
20     y))
21
22 # Create a GeoDataFrame from the list of random points
23 gdf_rand =
24
25 # Filter and prepare the countries GeoDataFrame
26 gdf_countries_filtered = gdf_countries[['name',
27
28
29 # Perform a spatial join to find which points fall within which
countries
30 gdf_joined =
31         gdf_countries_filtered,
32
33
34
35 # Count the number of points in each polygon (country)
36 gdf_counted =
37 gdf_counted =
38         columns = {'geometry' : 'number_of_points'})
```

39

```
40 # Display the count of points within the first 10 polygons (countries)
41 = 'number_of_points', ascending =
```

Out



In

```
1 # Merging the original GeoDataFrame with the counter DataFrame
2 # using the country names as joint keys
3 gdf_merged = left_on = 'name', right_index =
4 gdf_merged = gdf_merged[['name', 'number_of_points']]
5
```

Out



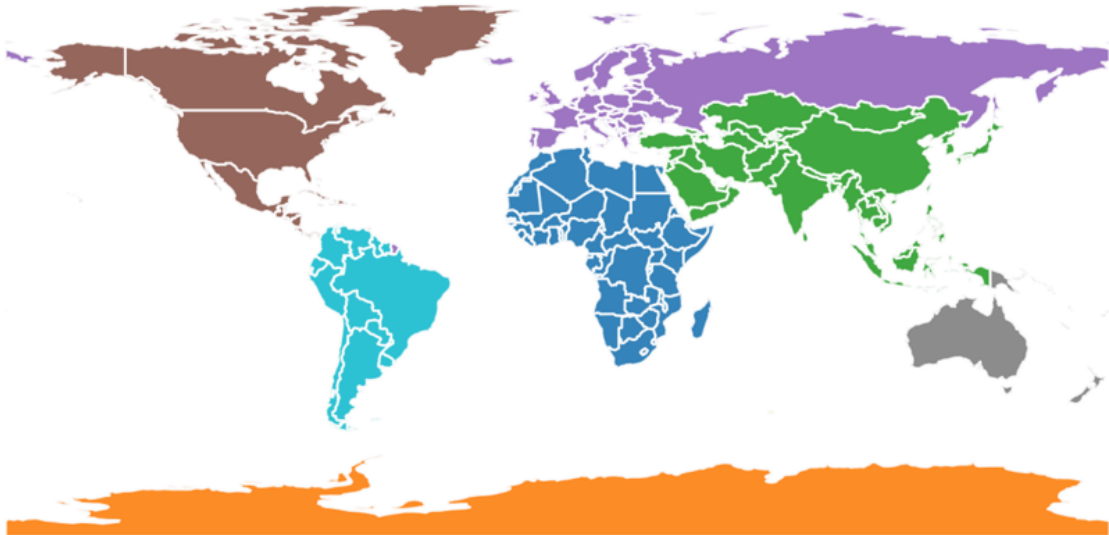
In this example, we combined several previously introduced steps to create new geospatial insights and merge them into an existing GeoDataFrame. By performing the counting points within polygons exercise, we can analyze spatial distributions and relationships, providing insights into patterns such as population density, event clustering, and resource allocation.

Summary on GeoPandas

In this chapter, we explored the comprehensive capabilities of GeoPandas, an extension of the Pandas library, designed specifically for geospatial vector data analysis. We began by understanding the difference between raster and vector data, and highlighting GeoPandas' strength in handling vector data through points, lines, and polygons. Via the Shapely - GeoPandas integration, we got a robust toolkit for various spatial operations, such as buffering, splitting, and merging geometries. We demonstrated practical applications, including creating GeoDataFrames from scratch, converting Pandas DataFrames to GeoDataFrames, and exporting GeoDataFrames to common GIS file formats like Shapefile and GeoJSON.

Furthermore, we explored tools for advanced spatial analysis, such as spatial joins and overlays, to combine and analyze different geospatial datasets. The chapter also covered methods for generating random synthetic data, performing point-in-polygon analyses, and visualizing geospatial data using GeoPandas' built-in plotting capabilities. Through practical examples and detailed explanations, we illustrated how GeoPandas can be used to derive meaningful insights from geospatial data, making it an indispensable tool for anyone working in spatial analytics.

Visualizing Geospatial Data



We may call geospatial data science the quantitative science of maps. As such, data visualization is a major pillar for understanding spatial information. Nowadays, there are a number of powerful techniques that transform raw spatial data into intuitive, interactive, and informative visual representations. In the sections that follow, we will explore the most widely used of these tools in Python.

First, we will learn how to use Matplotlib for both categorical and continuous data coloring and how to enhance our visualizations with context using base maps from Contextily. Then, we will make a detour to create interactive maps with Folium and Plotly. We will also cover advanced visualization techniques, such as generating heatmaps, overlaying multiple GeoDataFrames, and visualizing 3D geometries with Matplotlib and Pydeck. These examples will build the essential skills needed to create detailed, engaging, and insightful

geospatial visualizations that can be applied to a wide range of data analysis tasks.

31. Using Matplotlib for Categorical Coloring in Geospatial Data

Visualizing geospatial data with categorical values can provide clear and insightful representations of records of different kinds stored in GeoDataFrames. One effective way to achieve this is by using [Matplotlib](#) with categorical coloring.

In the following example, we will rely on the built-in world map dataset of `geopandas` and we will use Matplotlib to visualize a world map where each country is colored according to its continent stored in the categorical data column. First, we create a figure and axis using `plt.subplots` to ensure flexibility with the plot. In this example, we pick the figure size to be (10, 4), which measures the size of the canvas in the horizontal and vertical dimensions in inches.

Then, we introduced several further parameters for the plot function. The `alpha` parameter controls the transparency (with a value of 0.9, signaling 10% opacity), and `cmap` specifies the colormap (where I used the built-in `tab10` colormap). Additional formatting includes customizing the style of the boundary lines and adding a custom legend box to the plot.

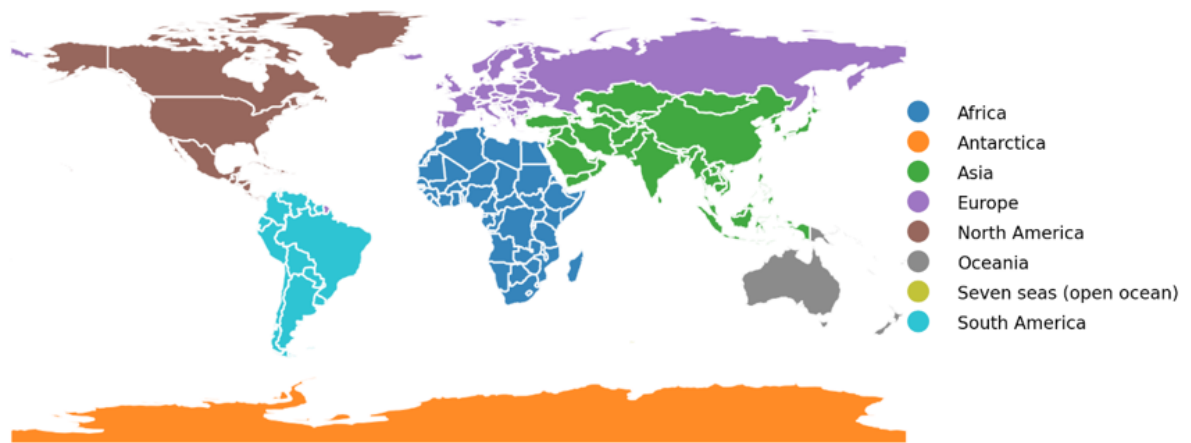
In

```
1 # Import the relevant packages
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4
5 # Load the sample dataset of world countries
6 gdf =
7
```



```
8 # Create a figure and axis for the plot
9 fig, ax = plt.subplots(1, 4))
10
11 # Plot the GeoDataFrame with categorical coloring by continent
12
13
14 # Column to determine colors
15 # Transparency level
16 # Colormap
17 # Color of the edges
18 # Width of the edges
19 # Display legend
20 )
21
22 # Customize the legend
23 legend = # Get the legend object
24 1)) # Position the legend outside the plot
25 # Turn off the legend frame
26
27 # Set font size for legend text
28 for text in
29
30
31 # Further adjust the legend position
32 0.8))
33
34 # Remove axis for a cleaner look

35
36
37 # Display the plot
38
```



This approach helps in creating clear and visually appealing maps that effectively represent different categories within our geospatial data. By using Matplotlib's extensive customization options, we can further tailor our maps to transform them into insights and data-driven stories.

32. Using Matplotlib for Continuous Value Coloring with Linear Scale

Visualizing geospatial data with continuous values can provide deep insights into the spatial variations and distributions within the data. One effective way to achieve this is by using Matplotlib with continuous value coloring, which will result in a so-called choropleth

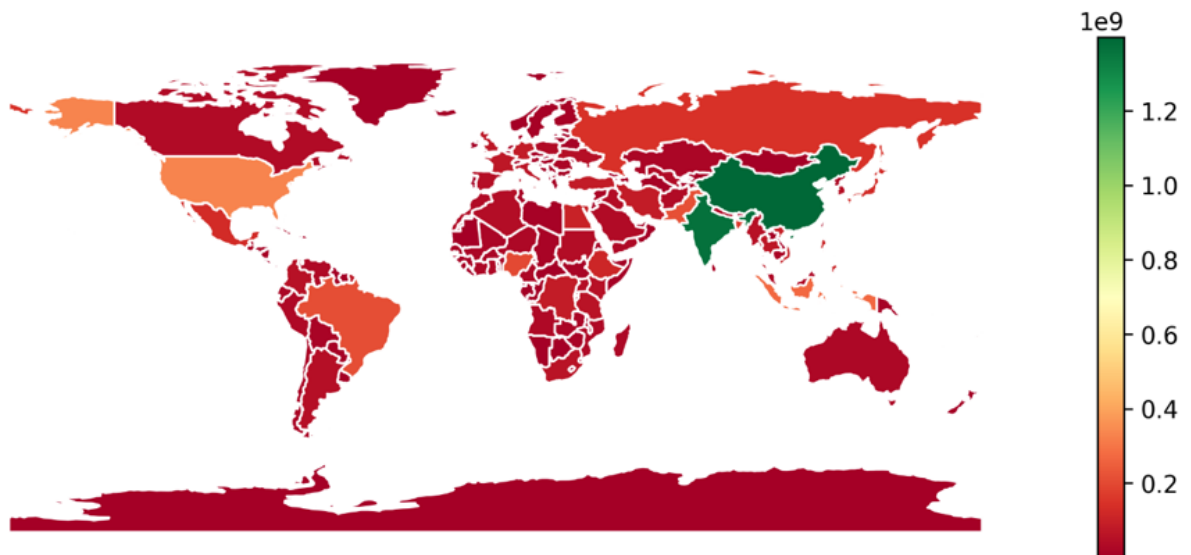
In the following example, we will again rely on the built-in world map dataset of GeoPandas, and we will use Matplotlib to visualize a world map where each country is colored according to its population estimate stored in the continuous data column. First, we create a figure and axis using `plt.subplots` to ensure flexibility with the plot. In this example, I picked the figure size to be (10, 4), which measures the size of the canvas in the horizontal and vertical dimensions in inches.

Then, we introduce several parameters for the plot function. The `column` parameter specifies the data column to visualize (in this case, `pop_est`), and `cmap` defines the colormap (here, we use the `RdYlGn` colormap, which transitions from red to yellow to green). The `linewidth` and `edgecolor` parameters control the appearance of the country borders, ensuring clear separation between countries. The `legend` parameter is set to `True` to display a legend, providing context for the color gradient representing the continuous values.

In

```
1 # Necessary package imports
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4
```

```
5 # Load the sample dataset of world countries
6 gdf =
7
8
9 # Create a figure and axis for the plot
10 fig, ax = plt.subplots(1, 4))
11
12
13 # Plot the GeoDataFrame with continuous value
14 # coloring by population estimate
15
16 # Column to determine colors
17
18 # Colormap
19 # Width of the edges
20 # Color of the edges
21 # Display legend
22 )
23
24
25 # Remove axis for a cleaner look
26
27
28
29 # Display the plot
30
```



This example creates a choropleth map, which is a type of map where areas are shaded or patterned in proportion to the value of a variable being represented. By using continuous value coloring, we can effectively visualize how different countries' population estimates compare across the globe, providing an insightful representation of the data.

As a closing note, one key observation from this map is that China and India are by far the greenest countries, with Brazil and the United States following in orange. Meanwhile, the vast majority of the map is colored red. This observation can be attributed to the distribution of country-level population values, which follows a heavily skewed power-law distribution. In the next topic, we will learn how to address this and improve our map visualization accordingly.

33. Using Matplotlib for Continuous Value Coloring with Logarithmic Scale

Visualizing geospatial data with continuous values can be further enhanced by applying different scaling techniques to better represent the data's distribution. One effective approach is to use a logarithmic which can provide more detail for data with a wide range of values. In this section, we will explore how to visualize geospatial data with continuous values using a logarithmic scale with the help of Matplotlib.

We will again rely on the built-in world map dataset of GeoPandas, and we will use Matplotlib to visualize a world map where each country is colored according to its population estimate stored in the continuous data column. This time, we will apply a logarithmic scale to the population data to better handle the wide range of values.

To carry out this scaling, first, we create a normalization object using `LogNorm` from Matplotlib, which scales the data logarithmically. We set the `vmin` and `vmax` parameters to the minimum and maximum values of the population estimates to define the range of the scale we introduced in the previous paragraph. Then, we create a figure and axis using the same `plt.subplots` with a figure size of (10, 4) inches.

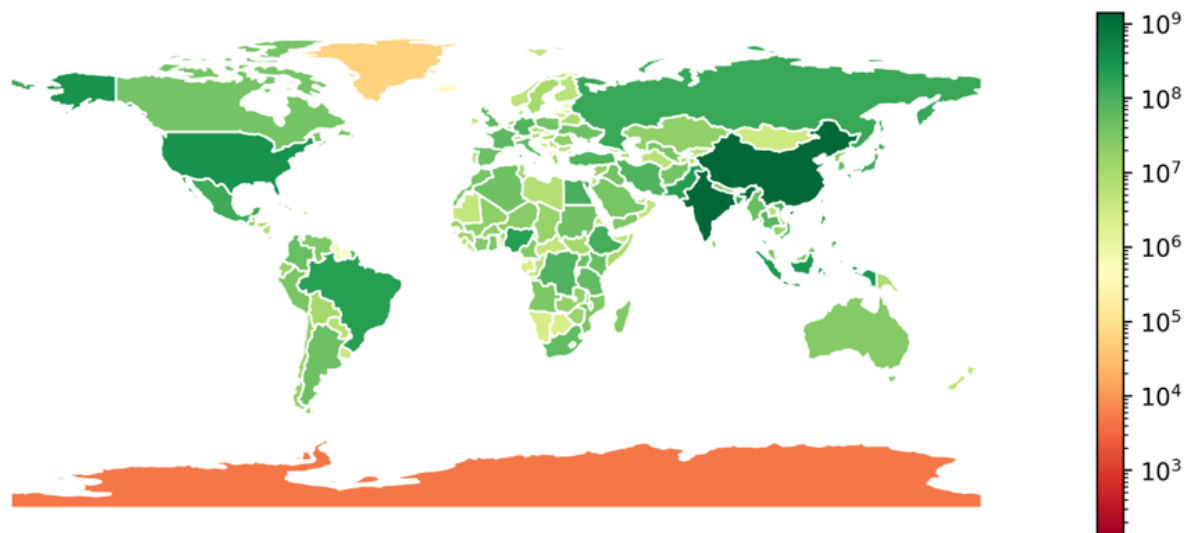
Finally, we use the `plot` function with several parameters: `column` specifies the data column to visualize, `cmap` defines the colormap and `norm` applies the logarithmic scale. The `linewidth` and `edgecolor` parameters control the appearance of the country borders, and the `legend` parameter is set to `True` to display a legend.

In

```
1 # Library imports
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4 from matplotlib.colors import LogNorm
5
6 # Load the sample dataset of world countries
7 gdf =
8
9 # Create a logarithmic normalization object
10 norm =
11
12 # Create a figure and axis for the plot
13 fig, ax = plt.subplots(1, 4)
14
15 # Plot the GeoDataFrame with continuous value coloring
16 # by population estimate using logarithmic scale
17
18 # Column to determine colors
19
20 # Colormap
21 # Width of the edges
22 # Color of the edges
23 # Display legend
24 # Apply logarithmic normalization
25 )
26
27
28 # Remove axis for a cleaner look
29
30
```

31 # Display the plot

32



Using a logarithmic scale helps to reveal patterns and details in the data that might be obscured by a linear scale, especially when dealing with data that spans several orders of magnitude, such as population levels of different geographies ranging from China to the Antarctic. This approach of scaling allows for a more nuanced and insightful representation of geospatial data, highlighting differences that are not immediately apparent with a linear scale.

34. Visualizing Multiple GeoDataFrames

Visualizing multiple GeoDataFrames on the same plot allows for the comparison of different datasets and the overlaying of additional information on a single map. This technique is particularly useful when we want to highlight specific features or compare various geographic datasets.

In the following example, we will use two GeoDataFrames: one representing a world map and another containing random geographic points we learned to generate in the previous chapter using `We will visualize the world map in a grey color and overlay the random points in crimson. This allows us to see the points in the context of the world map, enhancing our understanding of their geographic distribution.`

In the code block below, first we create a figure and axis using `plt.subplots` with a figure size of (10, 4) inches. Then, we plot the world map GeoDataFrame with the `plot` function, setting the `column` parameter to `continent` for categorization, `color` to `grey` for a neutral background, `alpha` for transparency, and `edgecolor` and `linewidth` for boundary styling. Next, we add the GeoDataFrame containing the random points by using the `plot` function, specifying the `markersize` and `color` parameters to highlight the points in `crimson`.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4 import numpy as np
```

```
5 from shapely.geometry import Point
6
7 # Load the sample dataset of world countries
8 gdf =
9
10 # Calculate the minimum and maximum bounds of the GeoDataFrame
11 bounds =
12 minx, miny, maxx, maxy = bounds
13
14 # Initialize an empty list to store random points
15 random_points = []
16
17 # Generate 1000 random points within the bounds
18 for _ in range(1000):
19     x = maxx
20     y = maxy
21     y))
22
23 # Create a GeoDataFrame from the list of random points
24 gdf_rand =
```

In

```
1 # Create a figure and axis for the plot
2 fig, ax = plt.subplots(1, 4)
3
4 # Plot the world map GeoDataFrame with grey color
5
6
```

7 # Column to determine colors

8 # Base color

9 # Transparency level

10 # Color of the edges

11 # Width of the edges

12)

13

14 # Overlay the random points GeoDataFrame with crimson color

15

16

17 # Size of the points

18 # Color of the points

19)

20

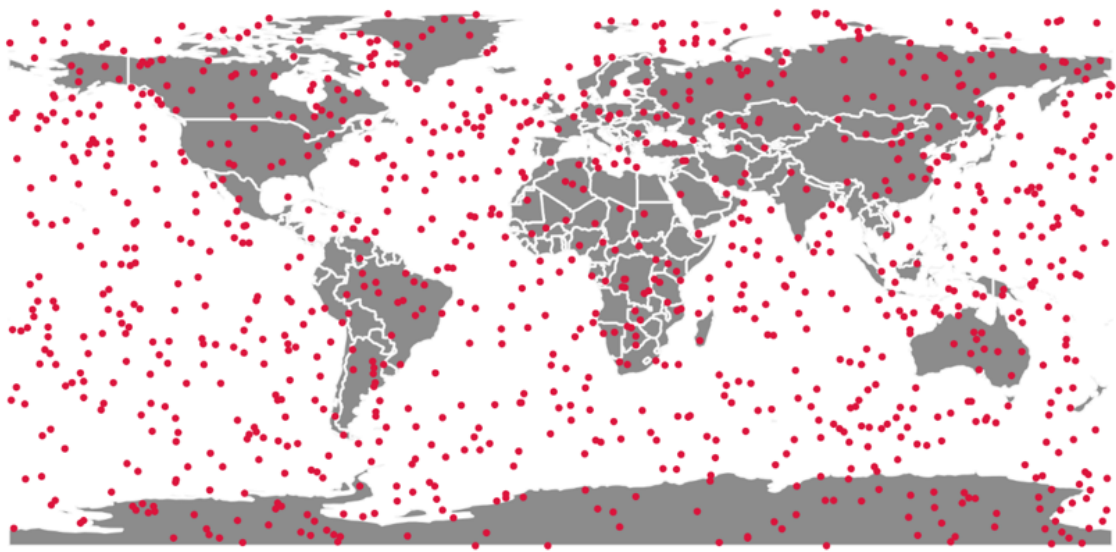
21 # Remove axis for a cleaner look

22

23

24 # Display the plot

25



By visualizing multiple GeoDataFrames on a single map, we can effectively combine and compare different datasets, enhancing the interpretability and richness of our geospatial visualizations. This technique is especially useful for overlaying additional data, such as points of interest, on a base map, providing a more comprehensive view of our geographic data.

35. Creating a Heatmap from Point Data

Heatmaps are an effective way to visualize measures such as the aggregates and density of point data on a map. They provide a clear representation of areas with high concentrations of points, which can be particularly useful for identifying hotspots and other spatial patterns within the data. In this section, we will explore how to create a heatmap from point data using the previously introduced Matplotlib and GeoPandas tools.

We will generate the usual set of random points within the bounds of a GeoDataFrame and create a heatmap to visualize the density of these points. We will first import the global map and then generate random points using Then we perform a spatial join between the random point GeoDataFrame to attach each point to its enclosing country (using within-tests in the background), and then we count the number of points within each country using the groupby function Finally, we plot the random points in crimson and visualize the heatmap using the count column to represent the density of points. We use the Reds colormap to create a gradient effect, highlighting areas with a higher number of points.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4 import numpy as np
5 from shapely.geometry import Point
6
7 # Load the sample dataset of world countries
```

```
8 gdf =
9
10 # Define the bounds of the GeoDataFrame
11 mins =
12 maxs =
13
14 # Generate random points within the bounds
15 random_points = []
16 for _ in range(1000):
17 x = maxs['maxx'])
18 y = maxs['maxy'])
19 y))
20
21 # Create a GeoDataFrame for the random points
22 gdf_rand =
23
```

Out



In

```
1 # Perform a spatial join to count the number
2 # of points within each geographic region

3 gdf_joined = gdf)
4 gdf_counted =
5
```

```
6 # Rename the 'geometry' column to 'count' and merge with the base
GeoDataFrame
7 gdf_counted =
8
9 gdf[['name', 'geometry']],
10
11 gdf_counted =
12
13 # Display the first 3 entries of the counted GeoDataFrame
14
```

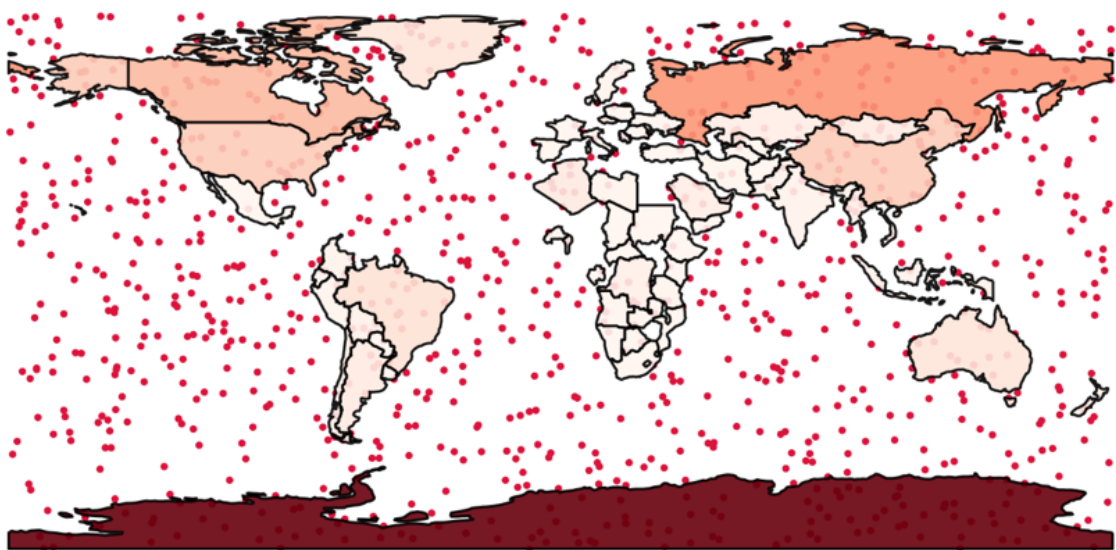
Out



In

```
1 # Create a figure and axis for the plot
2 fig, ax = plt.subplots(1, 4)
3
4 # Plot the random points in crimson
5
6
7 # Plot the heatmap using the 'count' column to represent density
8
9
10
11
```

```
12
13
14
15 # Remove axis for a cleaner look
16
17
18 # Display the plot
19
```



By combining random points with a world map we created a heatmap that effectively visualizes the number of data points within each country. As the points were scattered across the map randomly, we may note that the larger a country, the darker its color, serving as a quantitative indicator of area.

36. Adding Basemap with Contextily

Enhancing geospatial visualizations with base maps can provide valuable geographic context and make our data more informative and visually appealing. [Contextily](#) is a Python library that allows us to add base maps from various providers to our GeoPandas plots.

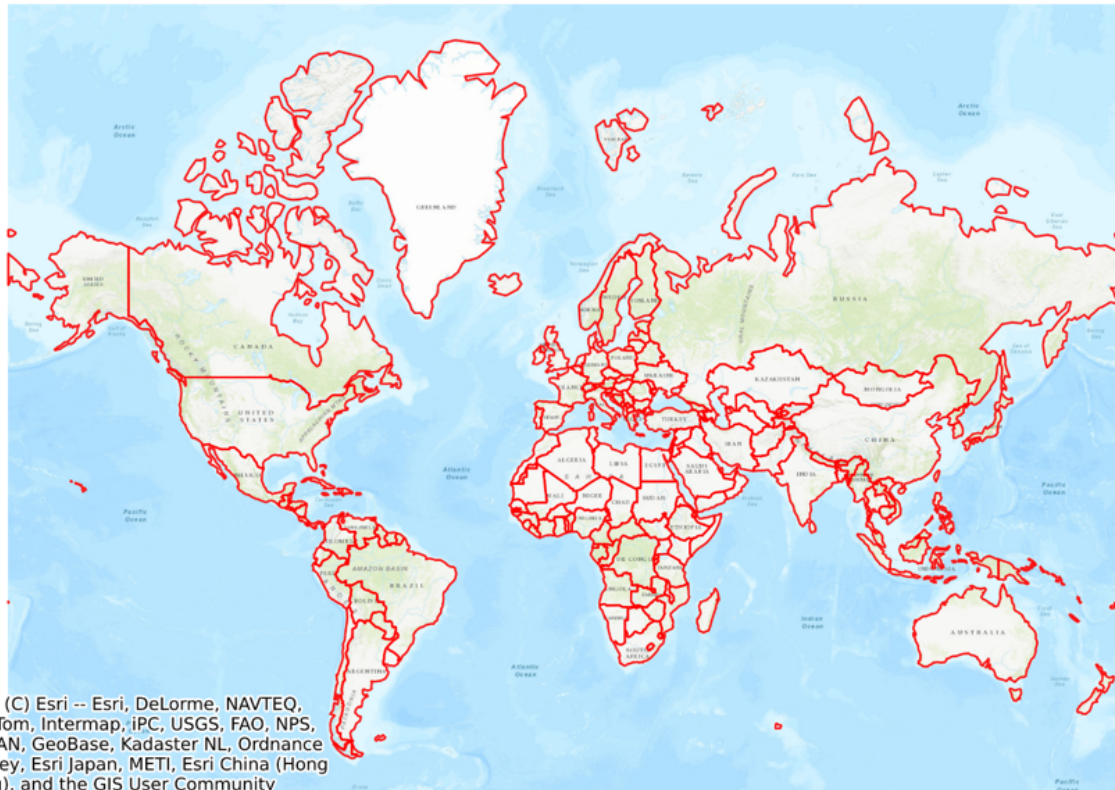
In the following example, we will use the usual GeoDataFrame (gdf) representing the world map and plot it on a basemap provided by Esri's When visualizing the map, we will adjust and customize the plotting parameters introduced earlier in this chapter, and project both the data set and the basemap to the Web Mercator projection. Additionally, to improve visibility, we discard Antarctica. This combined map will give us a detailed geographic background to better understand the context of the plotted data.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4 import contextily as ctx
5
6 # Load the sample dataset of world countries
7 gdf =
8
9 gdf = gdf[gdf.continent != 'Antarctica']
10 crs = 'EPSG:3857'
11 gdf = gdf.to_crs(crs)
12 # Create a figure and axis for the plot
```

```
13 fig, ax = 1, 4))
14
15 # Plot the GeoDataFrame with transparent fill and white edges
16
17
18 # Column to determine colors
19 # Transparency level
20 # Transparent fill
21 # Color of the edges
22 # Width of the edges
23 )
24
25 # Add basemap using Contextily
26
27 ax,
28 crs=crs, # Coordinate reference system of the GeoDataFrame
29 url=ctx.providers.Esri.WorldTopoMap # URL of the basemap provider
30 )
31
32 # Remove axis for a cleaner look
33
```

```
(np.float64(-22041259.17706817),
np.float64(22041259.177068174),
np.float64(-8777603.387860391),
np.float64(19736079.384779695))
```



Now, we also take a look at how to explore further base map providers and list their currently available base map types:

In

```
1 # Print the list of base map providers
2 providers =
3 print("Base map providers: ", dir(providers), "\n")
4
5 # Print the available base maps from Esri
6 esri_maps =
7 print("\nBase maps by Esri: ", dir(providers), "\n")
```

Base map providers: ['BasemapAT', 'CartoDB', 'Esri', 'FreeMapSK', 'GeoportailFrance', 'HERE', 'HikeBike', 'Hydda', 'JusticeMap', 'MapBox', 'MtbMap', 'NASAGIBS', 'NLS', 'OneMapSG', 'OpenFireMap', 'OpenMapSurfer', 'OpenPtMap', 'OpenRailwayMap', 'OpenSeaMap', 'OpenStreetMap', 'OpenTopoMap', 'OpenWeatherMap', 'SafeCast', 'Stamen', 'Thunderforest', 'Wikimedia', 'nlmaps']

Base maps by Esri: ['DeLorme', 'NatGeoWorldMap', 'OceanBasemap', 'WorldGrayCanvas', 'WorldImagery', 'WorldPhysical', 'WorldShadedRelief', 'WorldStreetMap', 'WorldTerrain', 'WorldTopoMap']

As we learned in this section, adding a base map, for instance, with Contextily, significantly enhances the visual context of our geospatial data via the detailed visual geographic background.

37. Creating a Simple Interactive Map with Folium

Interactive maps can provide a more engaging and informative way to visualize geospatial data by allowing us to explore the data sets in real-time. [Folium](#) is a powerful Python library that makes it easy to create such interactive maps by leveraging the Leaflet.js library.

In the following example, we will use the usual `GeoDataFrame` representing the world map and first convert it to GeoJSON format to ensure data compatibility with Folium using the `to_json` method. Then, we create a base map centered at the coordinates `[0, 0]` with an initial zoom level of 2 using the `folium.Map` function. We add the GeoJSON data to the map using the `folium.GeoJson` function. We also add tooltips to display the country names by specifying the `fields` and `aliases` parameters in

Finally, we add a layer control to the map to allow users to toggle different layers on and off using the `folium.LayerControl` function, save the interactive map as an interactive HTML file, and display it as the code cell's output.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 import folium
4
5
6 # Load the sample dataset of world countries
7 gdf =
8
```

```
9
10 # Convert GeoDataFrame to GeoJSON
11 geojson_data =
12
13
14 # Set the desired width and height in pixels
15 width = 800
16 height = 600
17
18
19 # Create a base map centered at coordinates [0, 0]
20 # with zoom level 2 and predefined size
21 m = 0],
22
23 # Add GeoJSON data to the map with tooltips for country names
24
25 geojson_data,
26
27
28
29
30 # Add layer control to the map
31
32
33 # Save the map to an HTML file
34
35
36 # Display the map in the notebook
37 m
```

Out

Make this Notebook Trusted to load map: File -> Trust Notebook



By using Folium, we can create interactive maps that allow users to explore geospatial data in a dynamic way. The addition of tooltips and layer controls enhances the user experience, making it easier to gain insights and understand the spatial relationships within the data.

38. Creating a Simple Interactive Map with Plotly

[Plotly](#) is a versatile Python library for creating interactive visualizations, including maps. By using Plotly, we can make interactive maps that allow for rich interactivity and detailed data exploration.

In the following example, we will use another built-in dataset of GeoPandas about global cities containing city locations worldwide. After a few preparatory steps, we convert the geometries to a pair of two-dimensional arrays and store them in the columns we call lon and lat.

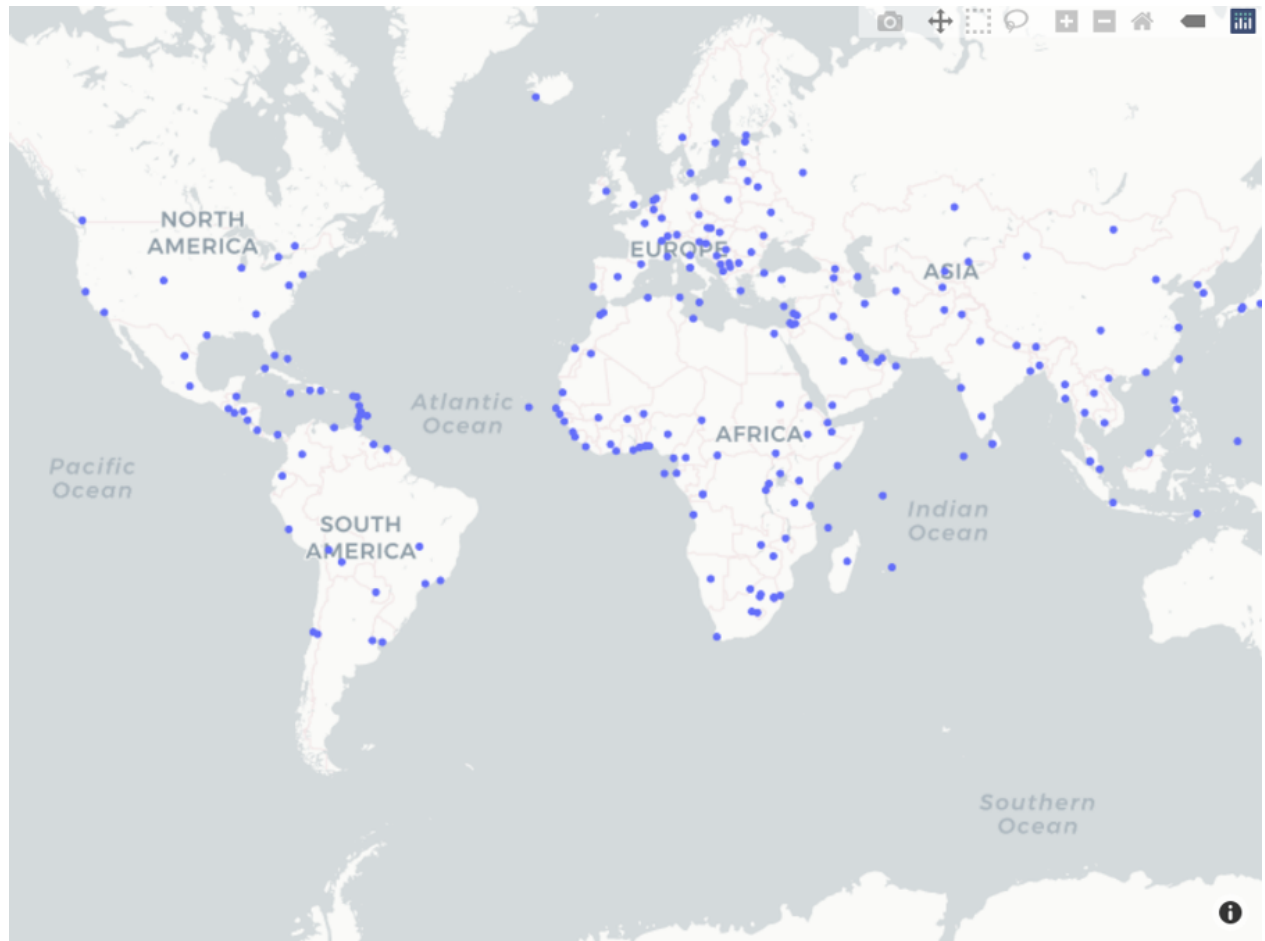
Then, we use the `px.scatter_mapbox` function from Plotly Express to create the interactive map. We specify the latitude and longitude columns we just created and include various formatting commands such as hover information, map style, zoom level, and center of the map. As a final touch, we update the layout to remove margins for a cleaner look and display the interactive map.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 import plotly.express as px
4
5 # Load the sample dataset of cities (replace with your actual data)
6 gdf_cities =
7
8 # Ensure the GeoDataFrame has a unique identifier column
9 gdf_cities['id'] =
```



```
10
11 # Step 1: Convert the geometries to a format Plotly can understand
12 gdf_cities['lon'] =
13 gdf_cities['lat'] =
14
15 # Step 2: Create a Plotly map
16 fig =
17 gdf_cities,
18
19
20
21
22
23 0, "lon": 0},
24 )
25
26
27 # Set the desired width and height in pixels
28 width = 800
29 height = 600
30
31 # Remove margins for a cleaner look and set the figure size
32
33
34
35
36 )
37
38 # Step 3: Show the figure
39 # fig.show()
```



By using Plotly, we can create highly interactive and visually appealing maps that are easy to integrate into web applications and interactive reports. The ability to add hover information and customize the map style enhances the user experience, making it easier to explore and understand the spatial aspects of our data.

39. Visualizing 3D Geometries with Matplotlib

While most of our geospatial journey has been covering 2D geometries, sometimes visualizing 3D geometries can provide a more comprehensive understanding of spatial relationships and structures within our data. Matplotlib's 3D plotting capabilities allow us to create and visualize complex 3D geometries created in Shapely, such as points, lines, and polygons.

In this section, we will briefly review how. In a few examples, we will use Shapely to create and plot 3D geometries, including a point, a line, and a polygon, and use the `mpl_toolkits.mplot3d` module from Matplotlib to create visuals.

In

```
1 # Import all necessary libraries
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4 from shapely.geometry import Point, LineString, Polygon
5
6 # Create 3D geometries
7 point_3d = Point(1.0, 2.0, 3.0)
8 line_3d = LineString([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0), (7.0, 8.0, 9.0)])
9 polygon_3d = Polygon([(0.0, 0.0, 0.0), (1.0, 0.0, 1.0), (1.0, 1.0, 2.0),
10 (0.0, 1.0, 3.0), (0.0, 0.0, 0.0)])
11
12 # Initialize the plot
13 fig =
14 ax =
```

15

16 # Plot the 3D point

17

18 Point')

19

20 # Plot the 3D LineString

21 line_x, line_y, line_z = p[1], p[2]) \

22 for p in

23 line_y, line_z,

24 LineString')

25

26 # Plot the 3D Polygon

27 poly_x, poly_y, poly_z = p[1], p[2]) \

28 for p in

29 poly_y, poly_z,

30 Polygon')

31

32 # Set labels

33

34

35

36

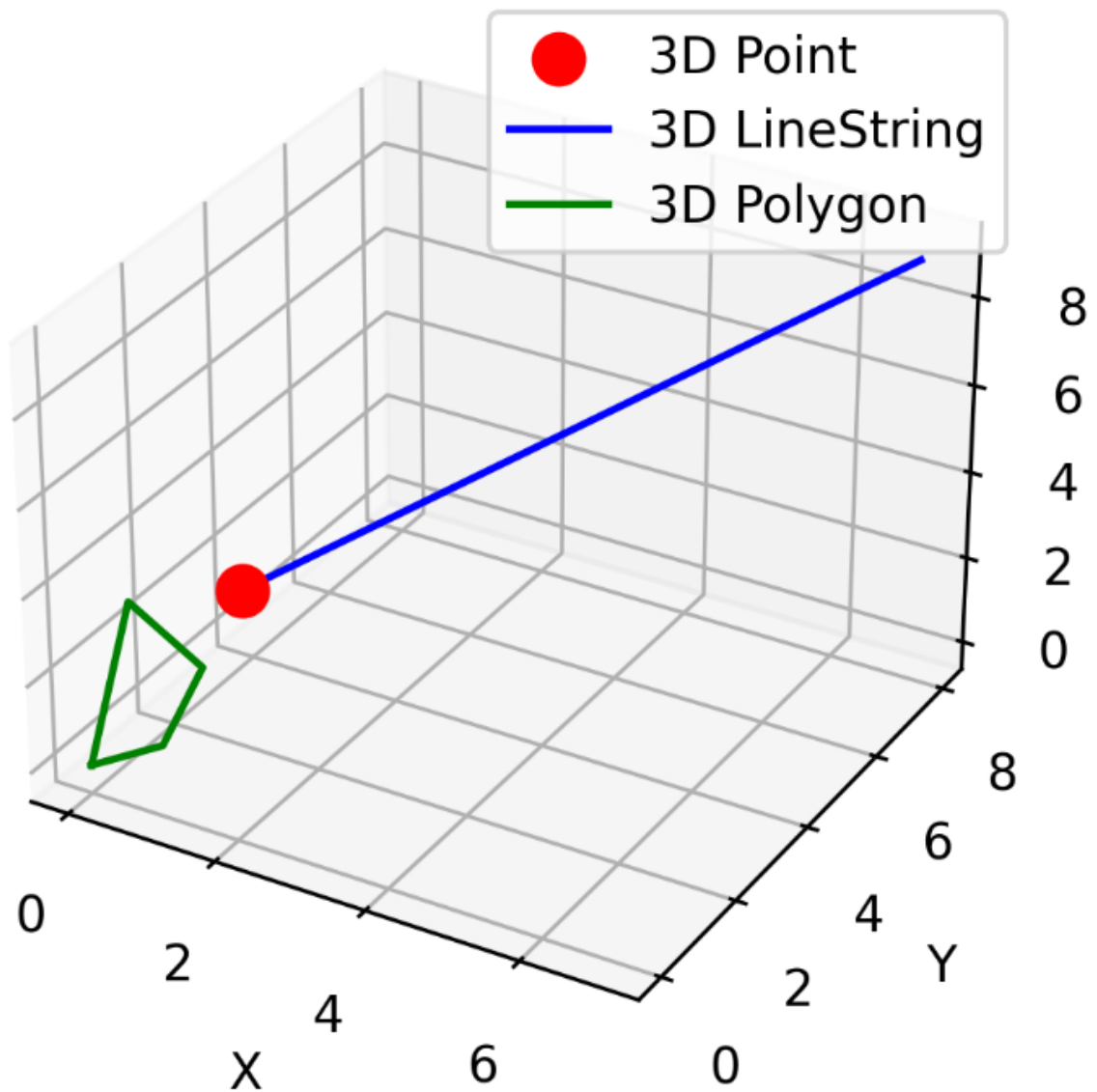
37 # Add legend

38

39

40 # Show the plot

41



Here, we show that we can use Matplotlib to create static, yet 3D visuals of geospatial data defined by Shapely.

40. Visualizing 3D Geometries with Pydeck

While Matplotlib is a straightforward choice for any Python-based visualization exercise, 3D visualizations sometimes demand more specific tools. One of those is a powerful library for creating interactive and visually appealing 3D visualizations. It leverages WebGL to render high-performance visualizations and is particularly well-suited for geospatial data.

In the following example, we will create a 3D polygon representing a column on a map using Shapely. Namely, we will extrude a 2D circle into a pillar, in other words, extending vertically to give it a 3D form at a specified height. While the height of the column is arbitrary, its location falls into one of the most iconic historical squares of Budapest - Liberty Square.

Once we have created the input GeoDataFrame, we move on to use Pydeck to render this geometry in 3D and incorporate interactive capabilities. First, we define a PolygonLayer in Pydeck, specifying the data source, polygon coordinates, and extrusion properties. The `get_elevation` property is used to set the height of the polygons, and we customize the appearance with color and highlight options.

Then, we set the initial view state for the map, including the latitude and longitude coordinates encoding the center of the map, as well as the default zoom level, and save the map as an HTML file. Finally, we also display the map on the code cell output.

In

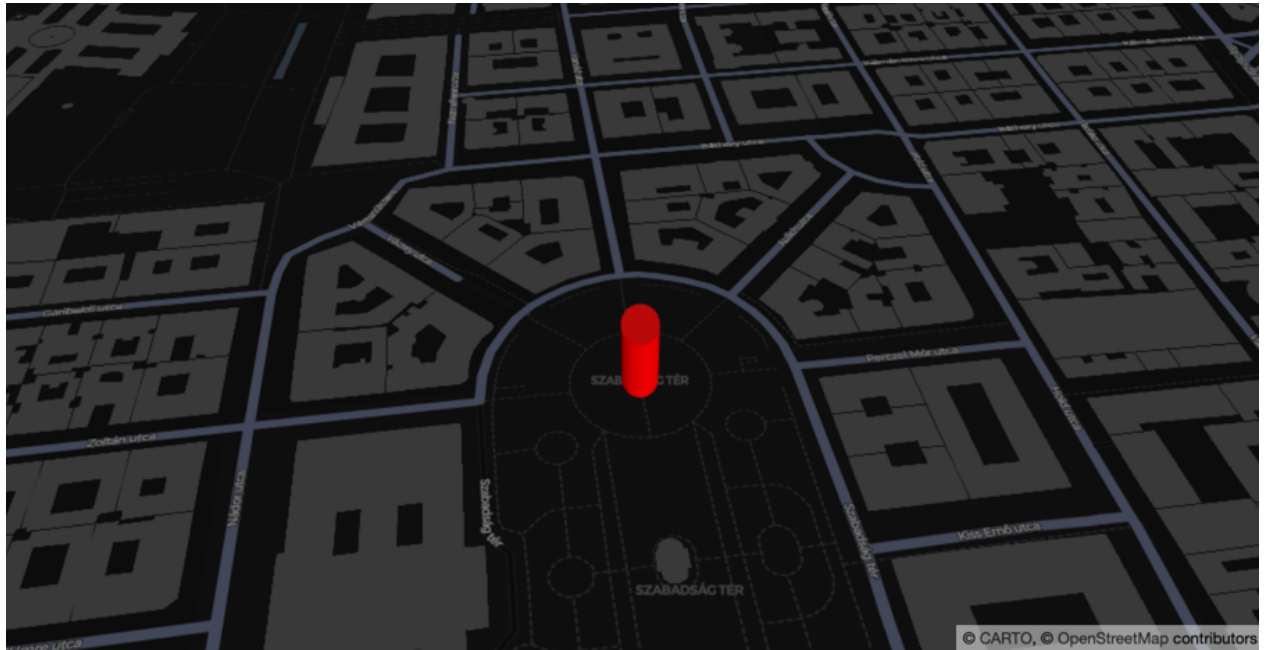
```
1 # Importing all necessary libraries
```

```
2 import geopandas as gpd
3 from shapely.geometry import Point
4 import pydeck as pdk

5
6 # Define coordinates for the column
7 x, y = 19.050331618793937, 47.50461561293986
8
9 # Create a 3D polygon geometry (column) and store it in a GeoDataFrame
10 column = Point(x,
11 gdf_column = column, 'height': 30}])
12
13
14
15 # Define the Pydeck layer
16 layer =
17 'PolygonLayer',
18
19
20
21
22
23
24
25 255, 255],
26 0, scaled * 255]"
27 )
28
29
30 # Set the initial view state
31 view_state =

32
```

```
33
34
35
36
37 )
38
39
40 # Set the desired width and height in pixels
41 width = 800
42 height = 600
43
44 # Create the 3D map with specified width and height
45 r =
46
47
48
49
50 # Display the map (uncomment the next line to save the map as an HTML
file)
51 # r.to_html('3d_column_map.html')
52 # r
```



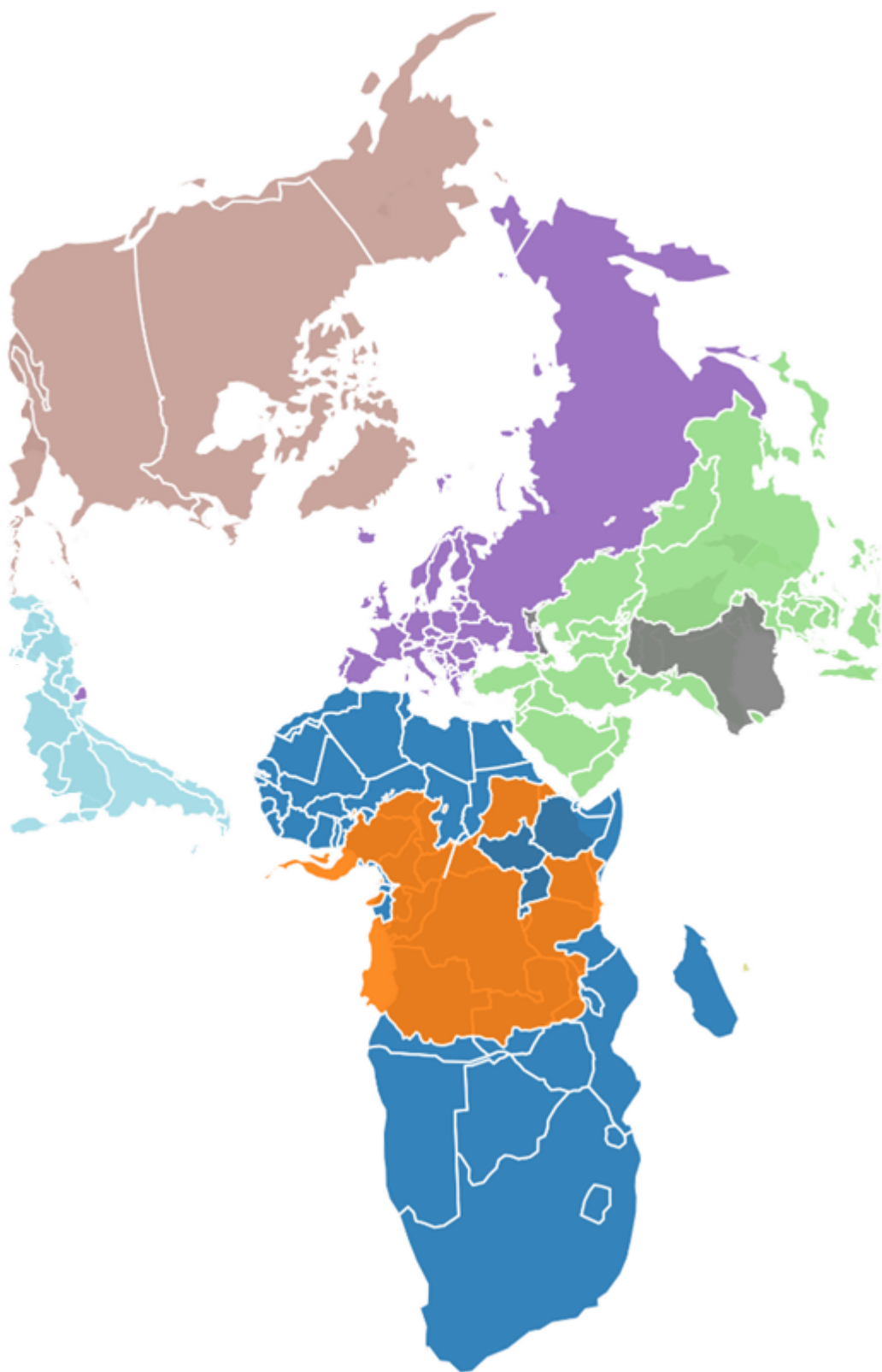
By using Pydeck, we can create interactive 3D visualizations that provide a rich and immersive experience for exploring geospatial data. This approach is particularly useful for visualizing complex three-dimensional spatial structures, such as building models and elevation maps.

Summary on Geospatial Data Visualization

In this chapter, we explored the diverse possibilities of visualizing geospatial data in Python, starting with the foundational tools to advanced interactive mapping libraries. We began by using Matplotlib for basic geospatial plots, including categorical and continuous value coloring. By leveraging Matplotlib's capabilities, we demonstrated how to create clear and insightful maps, using both linear and logarithmic scales. We also explored the creation of 3D geometries, showcasing how Matplotlib's 3D plotting capabilities can add depth and dimension to geospatial data, providing a richer context for analysis.

Advancing to more interactive and visually engaging tools, we introduced Folium, Plotly, and Pydeck. Folium and Plotly enabled us to create simple, interactive maps with ease, allowing users to explore geospatial data dynamically through a web-based interface. With Pydeck, we took geospatial visualization to the next level, rendering high-performance 3D maps that support complex geometries and real-time interaction. Through these examples, we illustrated the versatility and power of Python's geospatial visualization libraries, equipping readers with the skills to visualize and analyze spatial data across a wide range of applications, from urban planning to environmental monitoring.

Map Projections



In geospatial analytics, we study locations on the surface of the Earth, which is three-dimensional. However, most of our analytics, as well as our printed maps and computer screens, are in two dimensions. Map projection is the collection of techniques that allows us to project the three-dimensional surface onto two-dimensional planes tailored to the specific use case we are working on, from urban planning to naval navigation. Hence, understanding and working with map projections is crucial for accurate geospatial data analysis and visualization.

Map projections enable us to represent the three-dimensional surface of the Earth on a two-dimensional plane, making it possible to create maps and perform spatial analysis. However, these transformations usually introduce distortions in various aspects of spatial data, such as distances, directions, and areas. These distortions must be managed carefully to maintain the integrity of spatial data.

In this chapter, we will explore the technical aspects of map projections, encoded in so-called coordinate reference systems (CRS). We will begin by overviewing the concept of map projections and the importance of selecting the appropriate CRS for our data. We will learn how to query and set the default CRS, as well as how to reproject GeoDataFrames from one CRS to another. We will discuss the differences between global and local CRS and introduce a selection of global projection systems to broaden our understanding. By the end of this chapter, we will have a comprehensive overview of map projections and how to effectively manage CRS in our geospatial projects, ensuring that our spatial analyses and visualizations are both accurate and meaningful.

41. Querying Coordinate Reference Systems

Understanding the coordinate reference system (CRS) of our geospatial data is fundamental for accurate spatial analysis and visualization. The CRS defines how the two-dimensional, projected map in our GeoDataFrame is related to real places on the Earth. In this section, we overview three simple examples by querying and printing the CRS of three different built-in data sets of GeoPandas.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load sample dataset of world countries
5 gdf_countries =
6
7 # Load sample dataset of world world cities
8 gdf_cities =
9
10 # Load sample dataset of NYC borough boundaries
11 gdf_nyc =
```

In

```
1 # Query the CRS of each GeoDataFrame
```

```
2 print("CRS of world countries GeoDataFrame:")
3 "\n")
4
5 print("CRS of world cities GeoDataFrame:")

6 "\n")
7
8 print("CRS of NYC borough boundaries GeoDataFrame:")
9 "\n")
```

CRS of world countries GeoDataFrame:
epsg:4326

CRS of world cities GeoDataFrame:
epsg:4326

CRS of NYC borough boundaries GeoDataFrame:
epsg:2263

This code snippet will output the CRS information of each GeoDataFrame, allowing us to understand how our spatial data is projected. By knowing the CRS, we can ensure that different spatial datasets align correctly during analysis and visualization.

42. Setting the Default CRS

When creating a GeoDataFrame manually, it is crucial to set the coordinate reference system (CRS) correctly to ensure accurate spatial analysis and visualization. The CRS defines how the two-dimensional, projected map in our GeoDataFrame is related to real places on the Earth.

In this section, we will demonstrate how to set the default CRS of a manually created GeoDataFrame using GeoPandas. For this, we create a GeoDataFrame of cities where the cities' locations are described by their latitude and longitude coordinates. These coordinates, as shown in the previous example as well, belong to the EPSG:4326 CRS. The EPSG:4326 standard is commonly used for geographic coordinate systems and represents coordinates in degrees of latitude and longitude based on the WGS 84 datum. Here is a little more background information:

EPSG stands for the European Petroleum Survey Group, which provides a widely used standard for defining CRS codes. EPSG codes are numerical identifiers for specific CRS definitions, allowing for consistent referencing and transformation of spatial data.

Latitude and Longitude are the geographic coordinates used to specify locations on the Earth's surface. Latitude measures the distance north or south of the Equator, while longitude measures the distance east or west of the Prime Meridian.

WGS 84 (World Geodetic System 1984) is the standard geodetic datum used by the Global Positioning System (GPS). It provides a consistent frame of reference for measuring locations on the Earth's surface and is the basis for the EPSG:4326 CRS.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3 from shapely.geometry import Point
4
5 # Manually create a GeoDataFrame with city locations
6 data = {
7 'city': ['New York', 'Los Angeles', 'Chicago'],
8 'geometry': 40.7128),
9 34.0522),
10 41.8781)]
11 }
12 gdf_cities =
13
14 # Check the initial CRS of the GeoDataFrame (should be None)
15 print("There is no initial CRS of the GeoDataFrame:")
16 "\n")
17
18 # Set the CRS to EPSG:4326 (WGS 84)
19 = 4326
20
21 # Verify that the CRS has been set correctly
22 print("\nCRS after setting to EPSG:4326:")
```

23 "\n")

There is no initial CRS of the GeoDataFrame:

None

CRS after setting to EPSG:4326:

EPSG:4326

Additionally, we show the full information panel of a GeoDataFrame by simply outputting it within the following cell.

In

1

Out

2D CRS: EPSG:4326>

Name: WGS 84

Axis Info [ellipsoidal]:

- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)

Area of Use:

- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)

Datum: World Geodetic System 1984 ensemble

- Ellipsoid: WGS 84
- Prime Meridian: Greenwich

We may read the detailed CRS on the cell's output as follows. It describes the EPSG:4326, also known as WGS 84 (World Geodetic System 1984), which specifies locations using geodetic latitude and longitude, measured in degrees, with latitude ranging from -90° to 90° (south to north) and longitude from -180° to 180° (west to east). This system models the Earth's shape using the WGS 84 ellipsoid and uses the Greenwich Meridian as the prime meridian.

43. Reprojecting a GeoDataFrame

Reprojecting a GeoDataFrame is an essential task in geospatial analysis, as it allows us to transform our data into a different coordinate reference system (CRS). This transformation is necessary when we need to align datasets that use different projections or when a specific analysis requires a particular CRS.

In the following example, first, we load a sample dataset of world countries using GeoPandas' built-in dataset, which has WGS 84 (EPSG:4326) as a default CRS, also shown earlier. Then, we reproject the GeoDataFrame from its original CRS to EPSG:3857 (Web which is a popular projection used in web mapping applications, including Google Maps and OpenStreetMap). To do this, we will use the `to_crs` method of GeoPandas. Besides, we also output the details of the Web Mercator projection.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample dataset of world countries
5 gdf =
6
7 # Reproject the GeoDataFrame to EPSG:3857 (Web Mercator)
8 gdf_proj =
9
```

```
10 # Verify the new CRS of the reprojected GeoDataFrame
11 print("CRS before reprojecting to EPSG:3857:")
12 "\n")

13 print("CRS after reprojecting to EPSG:3857:")
14 "\n")
```

CRS before reprojecting to EPSG:3857:
epsg:4326

CRS after reprojecting to EPSG:3857:
EPSG:3857

In

1

Out

CRS: EPSG:3857>

Name: WGS 84 / Pseudo-Mercator

Axis Info [cartesian]:

- X[east]: Easting (metre)
- Y[north]: Northing (metre)

Area of Use:

- name: World between 85.06°S and 85.06°N.
- bounds: (-180.0, -85.06, 180.0, 85.06)

Coordinate Operation:

- name: Popular Visualisation Pseudo-Mercator
- method: Popular Visualisation Pseudo Mercator

Datum: World Geodetic System 1984 ensemble

- Ellipsoid: WGS 84
- Prime Meridian: Greenwich

As the cell output shows, we successfully reprojected the initial GeoDataFrame into EPSG:3857 CRS, which is detailed above. The description tells us that this projection system, also called as Pseudo-Mercator, utilizes a Cartesian coordinate system with eastings (X) and northings (Y) measured in meters. This projection is applicable worldwide between latitudes 85.06°S and 85.06°N , with bounds extending from -180° to 180° longitude.

44. Understanding Global vs. Local CRS

Selecting the appropriate coordinate reference system (CRS) is crucial in geospatial analysis, as the choice between a global or local CRS can significantly impact the accuracy and relevance of our results. A global CRS, such as EPSG:4326 (WGS 84), is suitable for global datasets and visualizations but can introduce distortions in distance and area calculations. Conversely, a local CRS is designed for more precise measurements in a specific region, making it ideal for tasks like area computation. In this section, we will demonstrate the differences between global and local CRSs and how to reproject a GeoDataFrame accordingly.

We will illustrate these differences through map visualizations and area computations, focusing on Hungary as an example. We will visualize the country-wide map of Hungary and the original world map using both the global CRS (EPSG:4326) and a local CRS (EPSG:23700) specific to Hungary. These comparisons will clearly show how each projection affects the representation of geographic features in different parts of the world, and how pronounced distortions can occur when an inappropriate CRS is used.

After the visual demonstration, we will move on to computing areas, as accurate area computation is a critical consideration when choosing between a global and local CRS. Area calculations in a global CRS like EPSG:4326 can be highly inaccurate due to the distortions inherent in global projections. Local CRSs, however, are designed to minimize these distortions in specific regions, making them suitable for precise area measurements. In our example, we will compute the area of Hungary in both global and local CRSs to highlight the importance of using an appropriate CRS for accurate measurements. This exercise underscores the necessity of selecting the right

CRS for geospatial analyses to ensure the integrity and reliability of the results. Let's see the code!

In

```
1 # Import the necessary library
2 import geopandas as gpd
3 import matplotlib.pyplot as plt # Import the plotting library
4
5 # Load a sample dataset of world countries
6 gdf =
7
8 # Define the local CRS for Hungary (EPSG:23700)
9 crs_hun = 23700
10 hungary = == 'Hungary']
11 hungary_hun =
12
```

Out

```
CRS: EPSG:23700>
Name: HD72 / EOVI
Axis Info [cartesian]:
- Y[east]: Easting (metre)
- X[north]: Northing (metre)
Area of Use:
- name: Hungary.
- bounds: (16.11, 45.74, 22.9, 48.58)
Coordinate Operation:
```

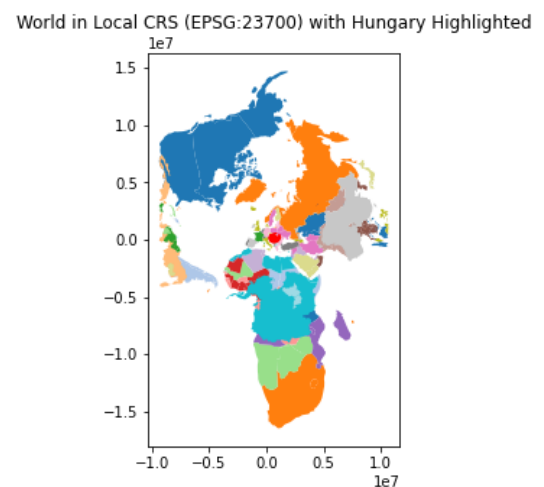
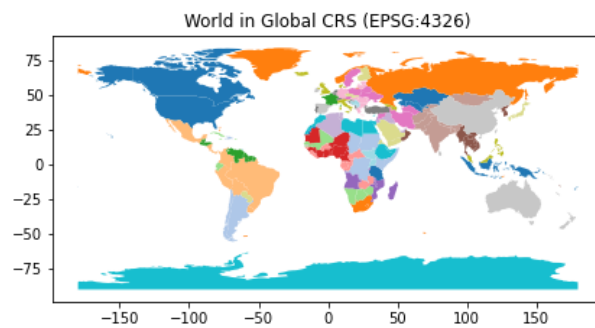
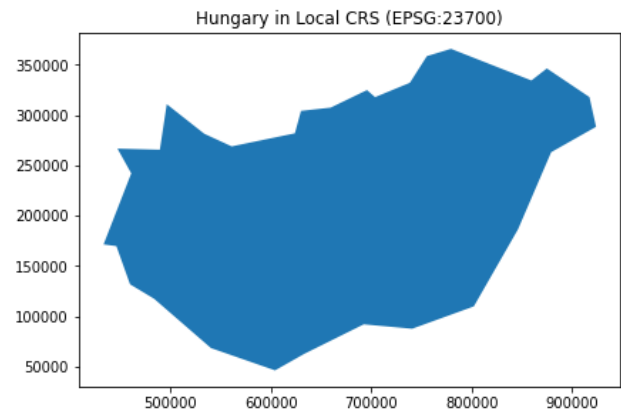
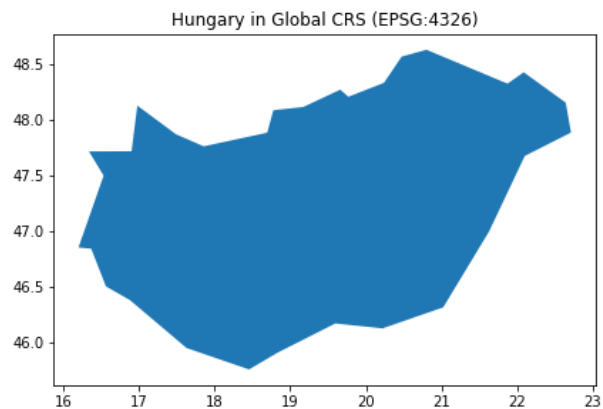

- name: Egyseges Orszagos Vetuleti
 - method: Hotine Oblique Mercator (variant B)
- Datum: Hungarian Datum 1972
- Ellipsoid: GRS 1967
 - Prime Meridian: Greenwich

In

```
1 # Plot Hungary in global and local CRS
2 fig, ax = plt.subplots(2, 1, figsize=(10, 5))
3
4
5 in Global CRS (EPSG:4326)
6 in Local CRS (EPSG:23700)
7
8 # Reproject the entire GeoDataFrame to the local CRS
9 gdf_hun = gdf_hun.to_crs('EPSG:23700')
10
11 # Plot the world in global CRS and Hungary in local CRS
12 fig, ax = plt.subplots(2, 1, figsize=(10, 5))
13 cmap = plt.cm.tab20
14 cmap = plt.cm.tab20
15
16 in Global CRS (EPSG:4326)
17 in Local CRS (EPSG:23700) with Hungary Highlighted)
```

Out

```
Text(0.5, 1.0, 'World in Local CRS (EPSG:23700) with Hungary Highlighted')
```



In

```

1 # Compute the area of Hungary in global CRS (EPSG:4326)
2 hungary['area_global'] =
3 print(f'Area of Hungary in global CRS (EPSG:4326): \
4     "\n")
5
6 # Compute the area of Hungary in local CRS (EPSG:23700)
7 hungary_hun['area_local'] =
8 print(f'Area of Hungary in local CRS (EPSG:23700): \
9     "\n")

```

Area of Hungary in global CRS (EPSG:4326): 10.980057532556389

Area of Hungary in local CRS (EPSG:23700): 92475472642.47691

The results of the previous code blocks illustrate some similarities but also striking differences between local and global CRS. First, we observe that using the local CRS for Hungary does not alter the shape of the country at all. However, a closer look at the axis labeling reveals a switch from longitude and latitude coordinates to SI units, with length measured in meters. This demonstrates the suitability of the local CRS for Hungary.

Conversely, when applying the local CRS of Hungary to the entire planet, the results are dramatically different. The map becomes distorted, centered around Hungary, and countries are positioned in completely unrecognizable places, highlighting the inadequacy of a local Hungarian CRS for global datasets.

The differences become even more pronounced when computing the area of Hungary. In EPSG:4326, the area calculation is ambiguous and inaccurate, yielding a value of around 10.98 in rather arbitrary units of measure. This value is rather far from the official 93,026 km² area of the country's official area, which, on the other hand, aligns well with the area computed using the local CRS.

This discrepancy underscores the inaccuracy of area calculations in a global CRS. In contrast, using a local CRS provides an area measurement much

closer to the official value, highlighting the importance of selecting the appropriate CRS for precise geospatial analysis.

45. Additional Projection Systems

Understanding and using different map projection systems is crucial for various geospatial applications. Each projection system has unique properties and uses, making it suitable for specific types of spatial analysis and visualization. In this section, we will explore the following additional projection systems, demonstrating how to apply them to a GeoDataFrame using GeoPandas and how to visualize a world map transformed into these CRS:

Eckert II: An equal-area pseudocylindrical projection used for world maps, known for reducing distortion of area and shape.

Equiarectangular: A simple equidistant cylindrical projection often used in raster data sets.

Lagrange: A rare projection system used in astronomy and celestial mapping.

Larrivee: A pseudocylindrical projection used for thematic maps.

Mollweide: An equal-area projection used for global maps, minimizing area distortion in exchange for shape distortions. Generally used for maps of the world or celestial sphere.

Natural Earth: A pseudocylindrical projection designed for world maps, balancing shape and area distortion.

In

```
1 # Import the necessary libraries
2 import geopandas as gpd #

3 import matplotlib.pyplot as plt
4
5 # Load a sample dataset of world countries
6 gdf =
7
8 # Define a dictionary of projection systems
9 # with their corresponding PROJ strings
10 projection_systems = {
11 'Eckert II': '+proj=eck2 +lon_0=0 +x_0=0 +y_0=0 \
12             +datum=WGS84 +units=m
13 'Equirectangular': '+proj=eqc +lon_0=0 +lat_ts=0 \
14                    +x_0=0 +y_0=0 +a=6378137 +b=6378137 \
15                    +units=m
16 'Lagrange': '+proj=lagrng',
17 'Larrivee': '+proj=larr',
18 'Mollweide': '+proj=moll +lon_0=0 +x_0=0 +y_0=0 \
19             +datum=WGS84 +units=m
20 'Natural Earth': '+proj=eqearth +lon_0=0 +x_0=0 +y_0=0 \
21                 +datum=WGS84 +units=m
22 }
23
24 # Apply each projection system to the GeoDataFrame and plot the
  results
25 for system_name, system_code in
26 gdf_proj =
```

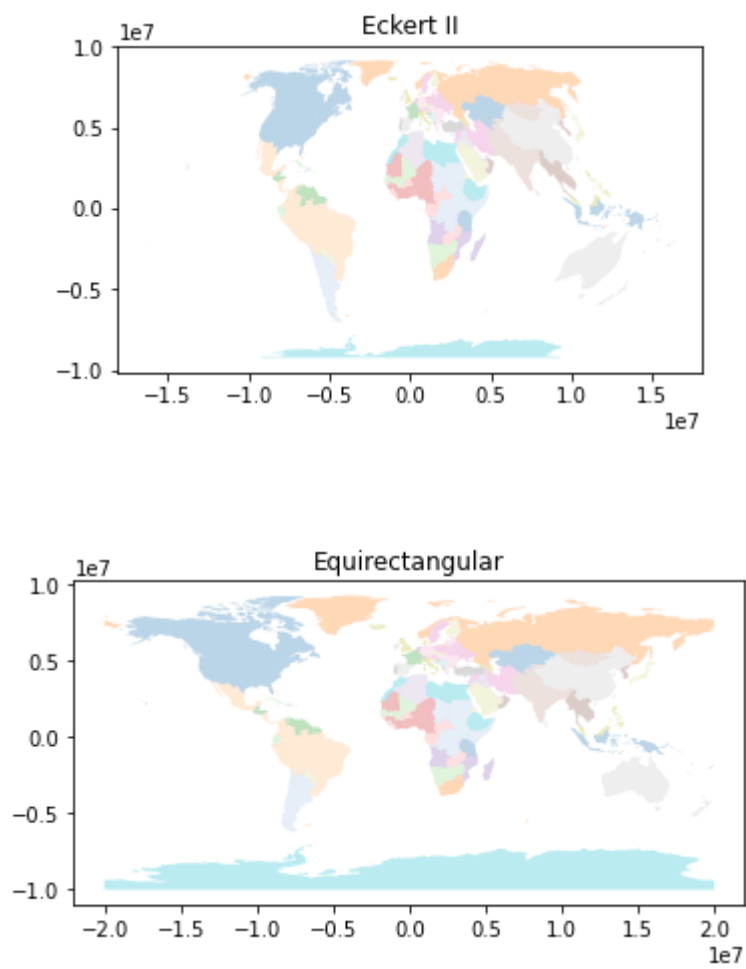
27 fig, ax = 1, 3))

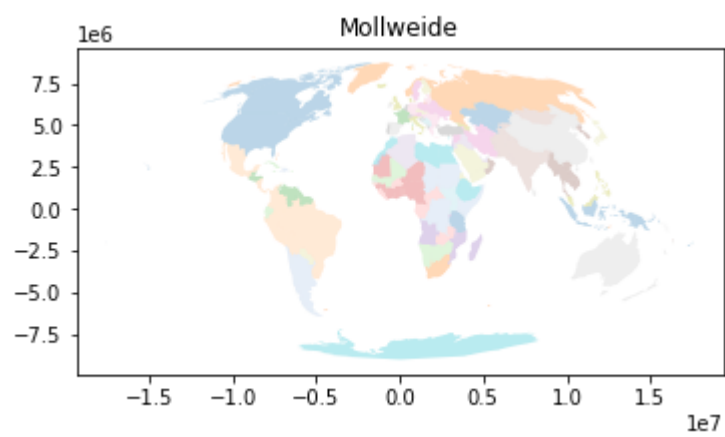
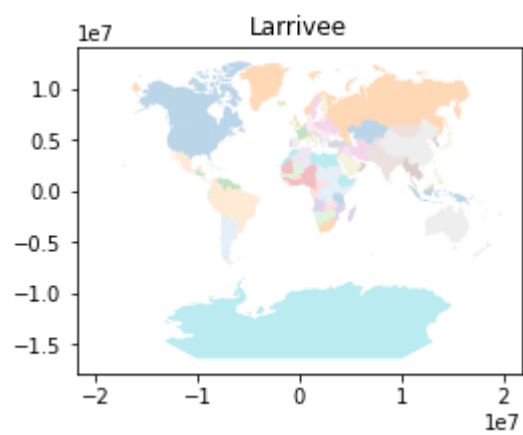
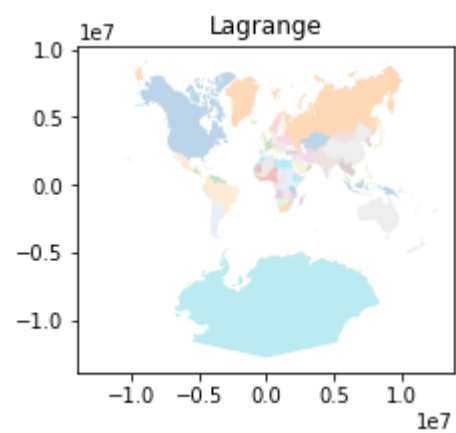
28

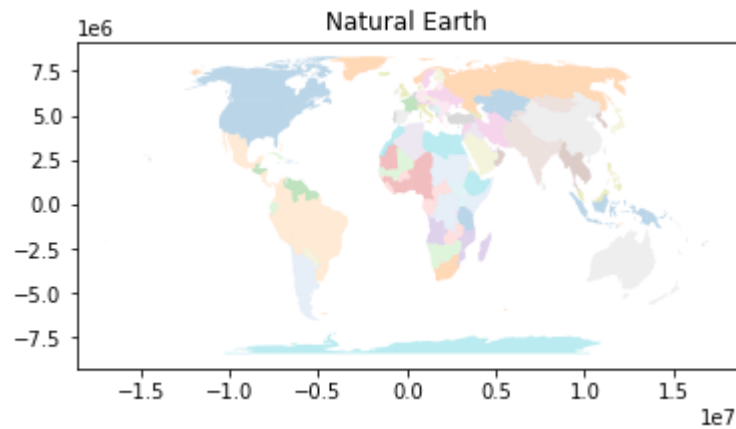
29

30

31







By exploring these (and further) additional projection systems, we can select the most appropriate projection for our specific geospatial analysis and visualization needs, ensuring accurate and meaningful representations of our spatial data.

46. Transforming Coordinates Directly

While we have already seen how to transform entire GeoDataFrames from one CRS to another, directly transforming coordinates between different coordinate reference systems (CRS) is just as much of a fundamental task in geospatial analysis. In this section, we will demonstrate how to transform coordinates using directly the [pyproj](#) library in Python.

For that purpose, we load the usual sample dataset of world countries using GeoPandas' built-in dataset. Then, as an example, we selected the records of Hungary and computed its centroid in the global CRS (EPSG:4326). In the next step, we extract the two coordinates from the centroid geometry and use the Transformer class from pyproj to convert coordinates from EPSG:4326 to EPSG:3857 and then from EPSG:3857 EPSG:23700.

In

```
1 import geopandas as gpd # Import the necessary library
2 from pyproj import Transformer # Import the Transformer class from
pyproj
3
4 # Load a sample dataset of world countries
5 gdf =
6
7 # Get the centroid of Hungary
8 hungary = gdf[gdf['name'] == 'Hungary']
9 print("Centroid of Hungary in global CRS (EPSG:4326):")
```

```
10 "\n")
11 centroid =
12
13 # Extract the latitude and longitude for a Hungary's centroid
14 lat =
15 lon =
16
17 # Create a transformer object to convert from EPSG:4326 to
EPSG:3857
18 transformer = "EPSG:3857")
19
20 # Transform the coordinates to EPSG:3858
21 x, y = lon)
22 print(f"\nOriginal coordinates (EPSG:4326): ({lat}, {lon})", "\n")
23 print(f"Transformed coordinates (EPSG:3857): ({x}, {y})", "\n")
```

Centroid of Hungary in global CRS (EPSG:4326):

```
115 POINT (19.35763 47.19995)
```

dtype: geometry

Original coordinates (EPSG:4326): (47.19995117195427,
19.357628627745918)

Transformed coordinates (EPSG:3857): (2154881.3618059703,
5974772.482175054)

In

```
1 # Create a transformer object to convert from EPSG:3857 to
  EPSG:23700
2 transformer = "EPSG:23700"
3
4 # Transform the EPSG:3857 coordinates to EPSG:23700
5 x2, y2 = y)
6 print(f"Original coordinates (EPSG:4326): ({x}, {y})", "\n")
7 print(f"Transformed coordinates (EPSG:23700): ({x2}, {y2})", "\n")
```

Original coordinates (EPSG:4326): (2154881.3618059703,
5974772.482175054)

Transformed coordinates (EPSG:23700): (673501.117446037,
206252.3436073569)

Directly transforming coordinates between different CRSs ensures that spatial data is accurately projected and aligned, even if not organized into higher-level data structures, such as a `GeoDataFrame`. By understanding how to use the `pyproj` library for coordinate transformation, we can now handle various geospatial data formats in a flexible way.

47. Obtaining EPSG Codes

EPSG codes, as we have seen across this chapter, are unique identifiers for different coordinate reference systems (CRS), making them essential for accurate geospatial data management and transformation. These codes ensure that spatial data can be correctly projected and analyzed. In this section, we will explore how to obtain EPSG codes specific to different locations using simple web scraping.

Online Resources for EPSG Codes

A comprehensive online database of coordinate reference systems. We can search for EPSG codes by location, name, or CRS properties. Here, we may browse EPSG codes manually as well.

[MapTiler](#) An API provided by MapTiler that allows us to search for coordinate reference systems by location. We need to create an account and obtain an API key to use this service.

Below I present an example of how to obtain EPSG codes using the MapTiler API. First, make sure to sign up for the [MapTiler API](#) and get our key after a simple free registration. Then, copy-paste our own key to the key variable and run the rest of the code cell. This code will use the [requests](#) library to make the API call and retrieve the JSON response from which we can extract and print the EPSG code information.

In

```
1 # Import the necessary library
2 import requests
3
4 # Replace with your MapTiler API key

5 key = 'your API key here>'
6 place = 'Hungary'
7
8 # Construct the API request URL
9 url = f'https://api.maptiler.com/coordinates/search/{place}.json?key=
{key}'
10
11 # Make the API request
12 response =
13 data =
14
15 # Print the first two results
16 print("First result for Hungary:")
17 print(data['results'][0], "\n")
18
19 print("\nSecond result for Hungary:")
20 print(data['results'][1], "\n")
```

First result for Hungary:

```
{'id': {'authority': 'ESRI', 'code': 37257}, 'kind': 'CRS-GEOGCRS', 'name':
'S-42 Hungary', 'exports': None, 'unit': None, 'accuracy': None, 'area':
'Hungary', 'bbox': [16.11, 45.74, 22.9, 48.58], 'deprecated': False,
'default_transformation': None, 'transformations': None}
```

Second result for Hungary:

```
{'id': {'authority': 'EPSG', 'code': 23700}, 'kind': 'CRS-PROJCRS', 'name':  
'HD72 / EOVI', 'exports': None, 'unit': 'metre', 'accuracy': 1.0, 'area':  
'Hungary.', 'bbox': [16.11, 45.74, 22.9, 48.58], 'deprecated': False,  
'default_transformation': {'authority': 'EPSG', 'code': 1242},  
'transformations': [1242, 1273, 1448, 1449, 1677, 1829, 1830, 1831,  
8183]}
```

In

```
1 # Repeat the process for New York  
2 place = 'New_York'  
3 url = f'https://api.maptiler.com/coordinates/search/{place}.json?key=  
{key}'  
4 response =  
5 data =  
6  
7 print("\nFirst result for New York:")  
8 print(data['results'][0], "\n")  
9  
10 print("\nSecond result for New York:")  
11 print(data['results'][1], "\n")
```

First result for New York:

```
{'id': {'authority': 'EPSG', 'code': 32118}, 'kind': 'CRS-PROJCRS', 'name':  
'NAD83 / New York Long Island', 'exports': None, 'unit': 'metre',
```

```
'accuracy': 4.0, 'area': 'United States (USA) - New York - counties of  
Bronx; Kings; Nassau; New York; Queens; Richmond; Suffolk.', 'bbox':  
[-74.26, 40.47, -71.8, 41.3], 'deprecated': False, 'default_transformation':  
{'authority': 'EPSG', 'code': 1188}, 'transformations': [1188, 1251, 1252,  
1308, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484,  
1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495, 1496,  
1497, 1498, 1499, 1500, 1501, 1502, 1503, 1515, 1520, 1521, 1522, 1523,  
1524, 1525, 1526, 1553, 1554, 1572, 1601, 1696, 1697, 1702, 1704, 1705,  
1706, 1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717,  
1718, 1719, 1720, 1721, 1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729,  
1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741,  
1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1752, 1843, 1848,  
1849, 1950, 8421, 8422, 8550, 8556, 8566, 8660, 8669, 8971, 9110, 9116,  
9117, 9118, 9119, 9241, 9243, 9244, 9550, 9795, 9887, 10013, 10014,  
10015, 10016, 10017, 10018, 10019, 10020, 15834, 15835, 15836, 15837,  
15838, 15839]]
```

Second result for New York:

```
{'id': {'authority': 'EPSG', 'code': 6538}, 'kind': 'CRS-PROJCRS', 'name':  
'NAD83(2011) / New York Long Island', 'exports': None, 'unit': 'metre',  
'accuracy': 2.0, 'area': 'United States (USA) - New York - counties of  
Bronx; Kings; Nassau; New York; Queens; Richmond; Suffolk.', 'bbox':  
[-74.26, 40.47, -71.8, 41.3], 'deprecated': False, 'default_transformation':  
{'authority': 'EPSG', 'code': 9774}, 'transformations': [9774]}
```

By using online resources and APIs like MapTiler, we can efficiently obtain EPSG codes for various locations, ensuring that our geospatial data is correctly referenced and projected. This capability is essential for accurate geospatial analysis and data management.

Summary on Map Projections

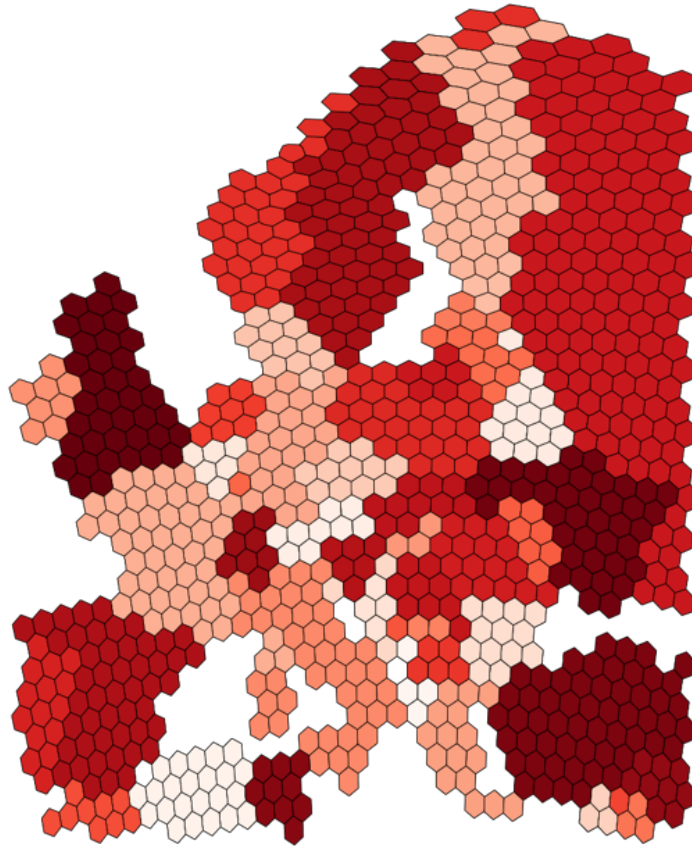
In this chapter, we reviewed the essential concepts and techniques related to map projections, which are fundamental for accurate geospatial analysis and visualization. We began by understanding the importance of coordinate reference systems (CRS) and how to query them for different GeoDataFrames.

We then explored how to set the default CRS for manually created GeoDataFrames and how to transform data into different CRSs to suit various analytical needs. We also examined the differences between global and local CRSs, using Hungary as an example to show how different projections impact spatial representations and measurements.

Furthermore, we introduced a variety of additional projection systems and covered the direct transformation of coordinates using the pyproj library, providing practical examples of how to convert between different CRSs for accurate spatial analysis. Lastly, we discussed methods for obtaining EPSG codes, utilizing online resources and APIs like MapTiler to find the appropriate CRS for different locations.

Through these sections, we equipped ourselves with a comprehensive understanding of map projections and CRS management, ensuring that our geospatial data analyses are accurate and meaningful. This knowledge is crucial for anyone working in fields such as urban planning, environmental science, and geographic information systems (GIS), where precise spatial data handling and the merging of different data sources is essential.

Spatial Indexing



As the amount of data we use grows by the day in every possible walk of life, managing and querying data efficiently becomes increasingly important in our daily work as data professionals. Geospatial data science and analytics is no exception. Spatial indexing serves as a powerful technique to enhance the performance of various spatial operations, enabling quick access and manipulation of even large-scale spatial data. By leveraging spatial indexes, we can perform complex queries and analyses with significantly reduced computational overhead.

Spatial indexing involves creating underlying data structures that enable efficient querying of spatial objects, such as points, lines, and polygons. These indexes improve the performance of spatial queries by organizing the geometries in a way that reduces the amount of data that needs to be examined during searches. Conversely, using such indices we can conduct rapid searching and retrieval of geographic features, making it possible to perform spatial joins, proximity searches, and other spatial operations more efficiently. In this chapter, we will explore various spatial indexing methods and their applications in geospatial analysis.

We will begin by demonstrating how to create a simple spatial index in GeoPandas, leveraging the built-in spatial index attribute to enhance query performance. We will then learn about using of RTree, a hierarchical spatial index structure that excels in handling dynamic spatial datasets. Finally, we explore square grids and the H3 hexagonal grids.

48. Creating a Simple Spatial Index in GeoPandas

Spatial indexing is a powerful technique in geospatial data analysis, allowing for efficient querying and management of spatial data. By creating spatial indexes, we can significantly speed up spatial queries by finding nearby features or performing spatial joins. GeoPandas provides built-in support for spatial indexing, making it easy to enhance the performance of our geospatial workflows.

GeoPandas' is a built-in spatial indexing mechanism. For now, we take that the `.sindex` attribute creates a spatial index for all the geometries stored in the given `GeoDataFrame`, allowing for rapid access to these indices when executing spatial operations such as intersection, containment, and proximity searches. By organizing spatial data into a structured index, `.sindex` significantly speeds up spatial queries.

As the example below shows, we demonstrate how to create a simple spatial index in GeoPandas. We will generate this spatial index for the sample, a `GeoDataFrame` of global countries, using the `sindex` attribute and print out the results.

In

```
1 # Import the necessary library
2 import geopandas as gpd
3
4 # Load a sample dataset of world countries
5 gdf = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
```

6

7 # Create a spatial index for the GeoDataFrame

8 `sindex = gdf.sindex`

9

10 # Print the details of the spatial index

11 `print("Type of spatial index:", type(sindex), "\n")`

12 `print("Spatial index details:", "\n")`

13 `print(sindex.__repr__(), "\n")`

Type of spatial index: 'geopandas.sindex.PyGEOSSTRTreeIndex'>

Spatial index details:

object at 0x7fd390177820>

The output of the code displays the type and details of the spatial index we created.

49. Using Simple Spatial Indexes for Efficient Queries

By leveraging spatial indexes, we can significantly speed up operations such as finding intersecting geometries and computing spatial joins. In this section, we will demonstrate how to use simple spatial indexes in GeoPandas to perform efficient queries, comparing the performance with and without using spatial indexes.

In the next example, we show how to narrow down the global map data set into a subset focusing on Europe. We will start by creating a spatial index for our GeoDataFrame of global countries using the `sindex` attribute, as shown in the previous section. Next, we define a bounding box that encompasses the majority of Europe using the `box` geometry function from Shapely. With the spatial index, we can quickly retrieve possible matches that intersect with the bounding box using the `intersection` method. Then, we refine our query by checking for exact geometry intersections within the possible matches to filter out any false positives.

To highlight the efficiency gained, we will compare the performance of this spatial indexing-based method with a simple, more direct approach relying on the `overlay` operation of GeoPandas. This comparison will illustrate the significant performance improvement in computational time provided by spatial indexing. Finally, for better explainability, we also plot the entire dataset, the overlaid component, and the query bounding box.

In

```
1 # Import the necessary library
```

```
2 import geopandas as gpd
3 from shapely.geometry import box
4 import time
5 import matplotlib.pyplot as plt
6
7 # Load a sample dataset of world countries
8 gdf = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
9
10 # Create a spatial index for the GeoDataFrame
11 sindex = gdf.sindex
12
13 # Define a bounding box for Europe
14 bbox = box(-10, 34, 40, 72)
15
16 # Perform a spatial query using the spatial index
17 # To identify all spatial indices that fall into the bounding box
18 t1 = time.time()
19 possible_matches_index = list(sindex.intersection(bbox.bounds))
20 possible_matches = gdf.iloc[possible_matches_index]
21 taken_for_spatial_index_query: {time.time() - t1} seconds", "\n")
22
23 # Refine the query using exact geometry intersection
24 precise_matches = possible_matches[possible_matches.intersects(bbox)]
25
26 # Print the number of possible and precise matches
27 print('Number of possible matches: ', len(possible_matches), "\n")
28 print('Number of precise matches: ', len(precise_matches), "\n")
29 time_taken_with_spatial_index: {time.time() - t1} seconds", "\n")
30
31 # Return the subset of the GeoDataFrame which are
32 #selected by the matching indices
33 subset_gdf = precise_matches
```

34

35 # Display the subset GeoDataFrame

36 display(subset_gdf.head())

Time taken for spatial index query: 0.003353118896484375 seconds

Number of possible matches: 48

Number of precise matches: 47

Total time taken with spatial index: 0.006163120269775391 seconds

| | |
|---------|---|
| seconds | seconds seconds seconds seconds seconds |
|---------|---|

| | | | |
|-----------------|-----------------|-----------------|-----------------|
| seconds seconds | seconds seconds | seconds seconds | seconds seconds |
| seconds seconds | seconds seconds | seconds seconds | seconds seconds |
| seconds | seconds | seconds | seconds |

In

1 # Comparison without spatial index by using GeoPandas'

2 # overlay function to get the overlaying part of Europe's bounding box

3 # and the full data set

4 t1 =

5 gdf_bbox =

6 gdf_over = gdf,

7 print('Number of matches with overlay: ', len(gdf_over), "\n")

8 time taken without spatial index: - t1} seconds", "\n")

Number of matches with overlay: 47

Total time taken without spatial index: 0.040920257568359375 seconds

In

1 # Plotting

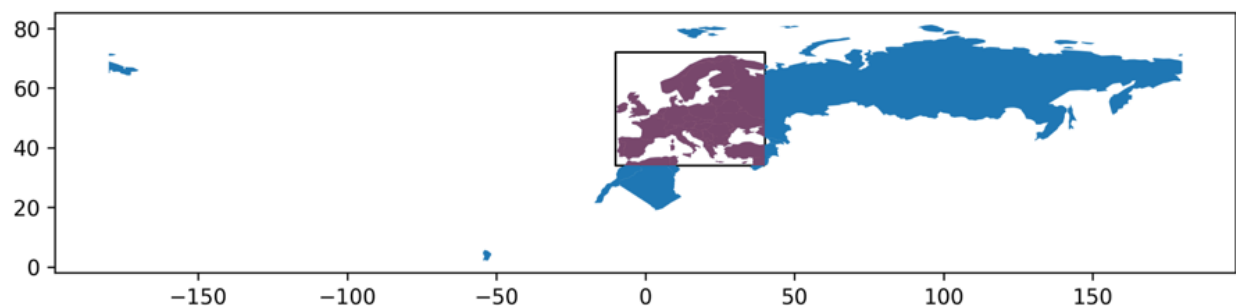
2 fig, ax = 1, 10))

3

4

5

6



The results printed by the previous code cells indicate a significant performance improvement when using a spatial index for spatial queries. The spatial index query on my computer took approximately 0.0033 seconds to identify 48 possible matches, and the precise intersection check refined this to 47 exact matches (country-level data records falling into the query box), completing the entire process in about 0.0061 seconds. In contrast,

performing the same task without a spatial index took about 0.041 seconds while yielding the same result of 47 matches. While the exact numbers may differ on different hardware, the spatial indexing leading to orders of magnitude faster results should remain.

50. Efficient Spatial Indexing with RTree

RTree is a powerful data structure primarily used for spatial indexing. RTree indexes are particularly useful for handling spatial data, as they enable quick searching of spatial objects such as points, lines, and polygons, and conducting spatial operations on them, such as finding points in polygons. In this section, we will demonstrate how to use RTree to create spatial indexes, insert geometries into the index, and perform spatial queries.

When taking a closer look, we need to remark that RTree is a hierarchical spatial index structure that organizes data in a tree of bounding rectangles, allowing for efficient querying of spatial objects. It dynamically adjusts as data is inserted or deleted, optimizing search operations. RTree is an effective tool for managing and querying spatial data in various geospatial analysis tasks. To use RTree in the next example, we rely on the [RTree](#) Python package.

In the following example, we will show how to use RTree to filter down a set of points on the plane to a subset of points falling within a given bounding box. We will start by creating an RTree index using the `index.Index()` constructor. Next, we will insert several `Point` geometries into the index. Each point will be assigned a unique identifier (index) and its bounding box will be used for insertion. We will then define a bounding box using the `box` function from Shapely, which will be used to query the RTree index. Then, we will query the index using the `intersection` method, which returns a list of indices of the points that intersect with the query bounding box. The results will be printed to show the points that lie within the query bounding box.

Finally, we use Matplotlib to visualize all the points located on the 2D plane, complemented by the query box and a detailed color coding.

In

```
1 # Import the R-tree index module
2 from rtree import index
3 from shapely.geometry import Point, box
4
5 # Create an R-tree index
6 idx =
7
8 # Insert some points into the index
9 points = [
10 Point(1, 1), Point(2, 2), Point(3, 3), Point(4, 4.1),
11 Point(1.5, 5), Point(1.5, 1.5), Point(2.0, 2.5), Point(4.5, 3.5),
12 Point(4.5, 4.9), Point(5.0, 5.5), Point(1, 2), Point(2, 3),
13 Point(3, 4.2), Point(4.1, 5), Point(5.5, 6)
14 ]
15
16 for i, point in enumerate(points):
17
18
19 # Create a bounding box to query
20 query_box = box(1.2, 1.3, 2.75, 2.85)
21
22 # Query the index
23 result =
24
25 # Output the results
26 print("Points within the query box:")
```

```
27 for i in result:
28 print(points[i])
```

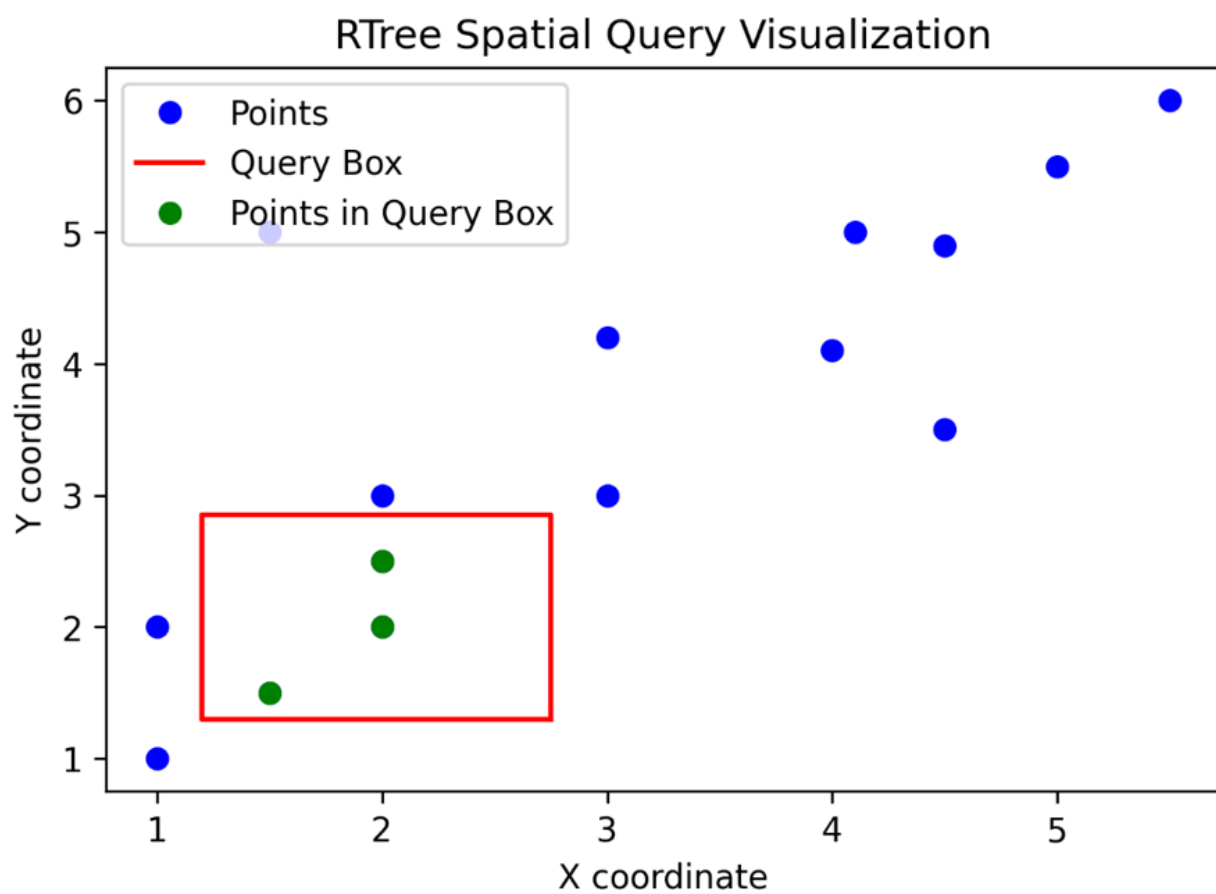
Points within the query box:

```
POINT (2 2)
POINT (1.5 1.5)
POINT (2 2.5)
```

In

```
1 # Import matplotlib for plotting
2 import matplotlib.pyplot as plt
3
4 # Visualization
5 fig, ax =
6 x, y = for point in points])
7 y, 'bo', # Plot the points
8
9 # Plot the query bounding box
10 x, y =
11 y, 'r-', Box')
12
13 # Highlight the points within the query bounding box
14 for idx, i in enumerate(result):
15 if idx == 0:
16 'go', in Query Box')
17 else:
18 'go')
19
20 # Add labels and legend
```

21 coordinate')
22 coordinate')
23 Spatial Query Visualization')
24 left')
25
26 # Show plot
27



The output will list the points from the initial set that fall within the defined query bounding box, also illustrated in the previous visualization.

51. Creating a Square Grid from Scratch

Creating a square grid is a simple geospatial task that can be used in various spatial analysis applications, from visualization to supporting efficient spatial querying by providing more controlled spatial indexing. Each manually created grid cell can serve as an index. Consequently, in this section, we will learn how to create a square grid from scratch.

We will start with a seed geometry, in this case, the administrative boundaries of Germany extracted from the built-in global map dataset. Then, we will define and implement a function called `create_grid` that generates grid polygons from a given input polygon. Specifically, this function takes the minimum and maximum x and y coordinates of the bounding box of the input polygon, as well as the number of cells we would like to produce along the x and y axes. After creating a 10x10 grid over Germany using this function, we will visualize the grid and the input administrative area polygon on the same figure using Matplotlib.

In

```
1 # Import GeoPandas for geospatial data handling
2 import geopandas as gpd
3 from shapely.geometry import box
4 import matplotlib.pyplot as plt
5
6 # Step 1: Create the grid polygons
7 def create_grid(minx, miny, maxx, maxy, num_cells_x, num_cells_y):
8     grid_polygons = []
```

```

9 cell_width = (maxx - minx) / num_cells_x

10 cell_height = (maxy - miny) / num_cells_y
11
12 for i in range(num_cells_x):
13 for j in range(num_cells_y):
14 x = minx + i * cell_width
15 y = miny + j * cell_height
16 y, x + cell_width, y + cell_height))
17
18 return grid_polygons
19
20 # Extract the bounding box of Germany from the GeoDataFrame
21 gdf =
22 gdf_example[ == 'Germany']
23 bounds =
24
25 minx =
26 miny =
27 maxx =
28 maxy =
29
30 # Create a 10x10 grid within the bounding box of Germany
31 grid_polygons = create_grid(minx, miny, maxx, maxy, 10, 10)
32 gdf_grid =
33
34

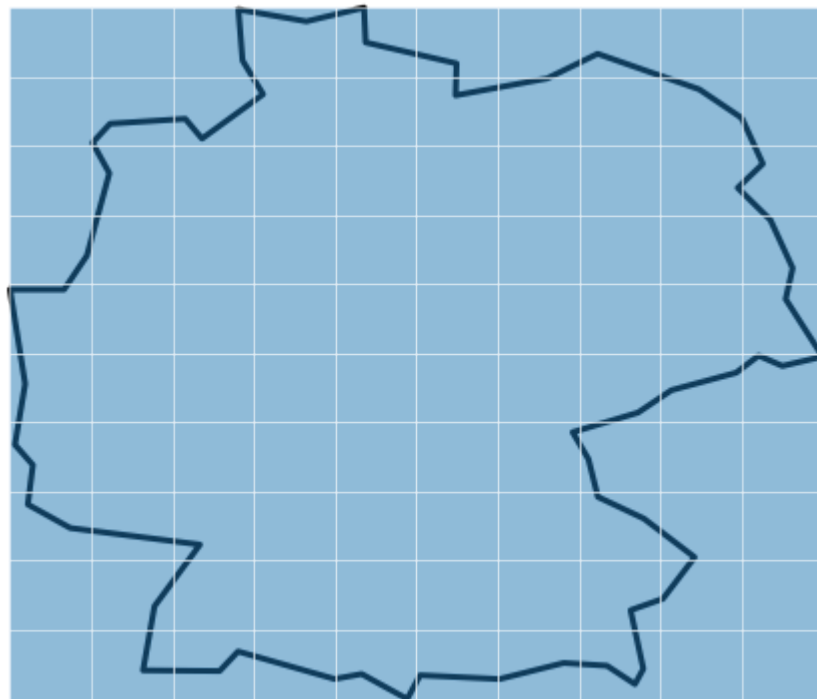
```

Out



In

```
1 # Visualize the grid overlaid on the map of Germany
2 f, ax = plt.subplots(1, 1, figsize=(8, 8))
3 gdf_example.plot(ax=ax, color='none', edgecolor='k', linewidth=3)
4 gdf_grid.plot(ax=ax, edgecolor='w', alpha=0.5)
5 ax.axis('off')
6 plt.show()
```



This example shows how to create a square grid over a specified region, in this case, Germany. The grid is created by dividing the bounding box of

Germany into equal-sized cells, and each cell is represented as a polygon. The visualization shows the grid and the map of Germany. This grid can be used for various spatial analyses, such as aggregating data points within each cell or performing spatial joins with other datasets.

52. Visualizing RTree Indexing

Following our previous example of creating a square grid, we can use this grid for more advanced spatial operations, such as incorporating an RTree index and visualizing the overlap between the bounding box and the input polygon.

To create the grid, we will use the `create_grid` function defined in the previous section, which generates grid polygons within a bounding box, for instance, that of a given input polygon. In this example, we will use the administrative boundaries of Germany again, extracted from the global map data set. Next, we use the `create_grid` function to create a 10x10 grid covering Germany's bounding box.

We then insert the grid polygons into an RTree index and create an indexed GeoDataFrame to store these polygons along with their indices. After this, we query the RTree index to find which grid cells intersect with Germany's geometries. We refine the results to get precise matches and mark intersecting cells. Finally, we visualize the grid overlaid on the map of Germany using Matplotlib, highlighting the intersecting cells.

In

```
1 # Import all used libraries
2 import geopandas as gpd
3 from shapely.geometry import box
4 from rtree import index
5 import matplotlib.pyplot as plt
```

6

7

8 # Extract the bounding box of Germany from the GeoDataFrame

9 gdf = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))

10 gdf_example = gdf[gdf.name == 'Germany']

11 bounds = gdf_example.bounds

12

13 minx = bounds.minx.values[0]

14 miny = bounds.miny.values[0]

15 maxx = bounds.maxx.values[0]

16 maxy = bounds.maxy.values[0]

17

18 # Create a 10x10 grid within the bounding box of Germany

19 grid_polygons = create_grid(minx, miny, maxx, maxy, 10, 10)

20 grid_polygons[0]

Out



In

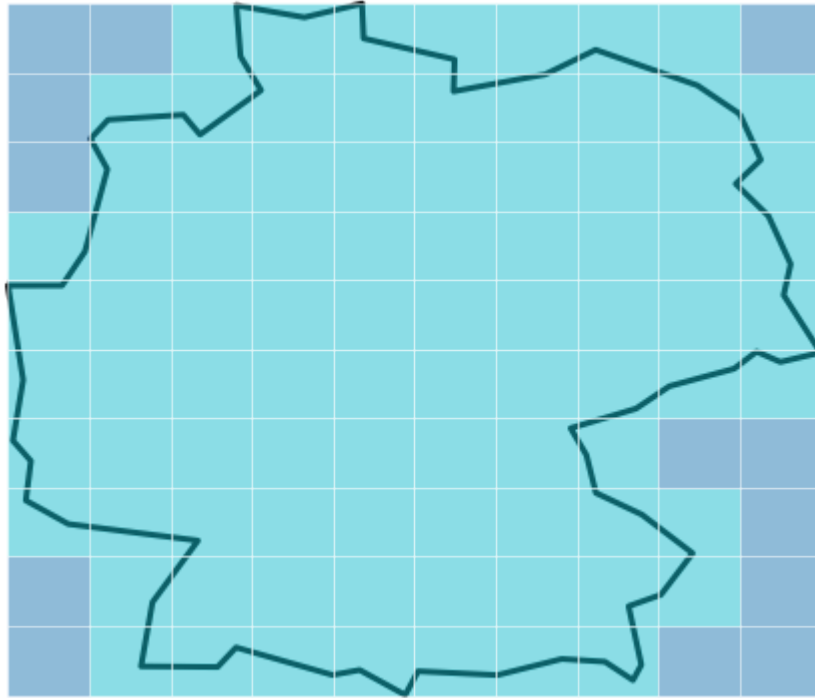
```
1 # Insert the grid polygons into an R-tree and create an indexed GDF
2 idx = index.Index()
3 grid_data = []
4 for i, poly in enumerate(grid_polygons):
5     idx.insert(i, poly.bounds)
6     grid_data.append({'geometry': poly, 'index': i})
7
8 gdf_grid = gpd.GeoDataFrame(grid_data, columns=['geometry', 'index'])
9
10 # Find which grid cells intersect with gdf_example
11 intersecting_indices = []
12
13 for poly in gdf_example.geometry:
14     possible_matches_index = list(idx.intersection(poly.bounds))
15     possible_matches = gdf_grid.iloc[possible_matches_index]
16     precise_matches =
possible_matches[possible_matches.intersects(poly)]
17
18 # Remove duplicates
19 intersecting_indices = list(set(intersecting_indices))
20
21 # Add a column to indicate if the grid cell intersects with gdf_example
22 gdf_grid['intersects'] = gdf_grid['index'].apply(\
23     lambda x: x in intersecting_indices)
24 gdf_grid.head(5)
```

Out



In

```
1 # Visualize the grid polygons
2 f, ax = plt.subplots(1, 1, figsize=(8, 8))
3
4 # Plot the example GeoDataFrame
5 gdf_example.plot(ax=ax, color='none', edgecolor='k', linewidth=3)
6
7 # Plot the grid with intersecting cells highlighted
8 gdf_grid.plot(column='intersects', ax=ax, edgecolor='w', alpha=0.5)
9 ax.axis('off')
10 plt.show()
```



This example shows the RTree indexing in action by creating a grid, indexing the grid cells, and querying the index to find intersecting cells between the bounding box and the country polygon.

53. Enclosing Grid Cell Identification Using RTree

Here, we look a little bit under the hood of spatial indexing and demonstrate how to identify the nearest enclosing grid cell for a given point using RTree indexing. This method is useful in various spatial analysis tasks where we need to find the grid cell that contains or is closest to a specific point. The process involves defining a point of interest, finding the nearest grid cell using RTree, and visualizing the point and its nearest grid cell.

To perform this task, we start with preparation tasks, such as loading the global country map and filtering the target data set to Germany. We also reuse the `create_grid` function and create a square grid from Germany's bounding box. Then, we define a random point manually picked from Germany using the `Point` class from Shapely.

Next, we add the RTree index. We then use the `nearest` method of the RTree index to find the nearest grid cell to a given point. The `nearest` method returns the index of the closest geometry, which, in this case, is the enclosing grid cell. After identifying the nearest grid cell, we visualize the point and the grid on a map using Matplotlib, highlighting the nearest cell in red.

In

```
1 # Import all the necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Point, box
4 from rtree import index
5 import matplotlib.pyplot as plt
```


6

7

```
8 # Extract the bounding box of Germany from the GeoDataFrame
9 gdf = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
10 gdf_example = gdf[gdf.name == 'Germany']
11 bounds = gdf_example.bounds
12
13 minx = bounds.minx.values[0]
14 miny = bounds.miny.values[0]
15 maxx = bounds.maxx.values[0]
16 maxy = bounds.maxy.values[0]
17
18 # Create a 10x10 grid within the bounding box of Germany
19 grid_polygons = create_grid(minx, miny, maxx, maxy, 10, 10)
```

In

```
1 # Define the point of interest
2 point = Point(13.01969, 52.05703)
3 print(point, "\n")
4 point
```

POINT (13.01969 52.05703)

Out



In

```
1 # Insert the grid polygons into an R-tree and create a GeoDataFrame
2 idx = index.Index()
3 grid_data = []
4 for i, poly in enumerate(grid_polygons):
5     idx.insert(i, poly.bounds)
6     grid_data.append({'geometry': poly, 'index': i})
7
```

```
8 gdf_grid = gpd.GeoDataFrame(grid_data, columns=['geometry', 'index'])
9
10
11
12 # Find the nearest grid cell to the point of interest
13 nearest = list(idx.nearest(point.bounds, 1))[0]
14 nearest
```

Out

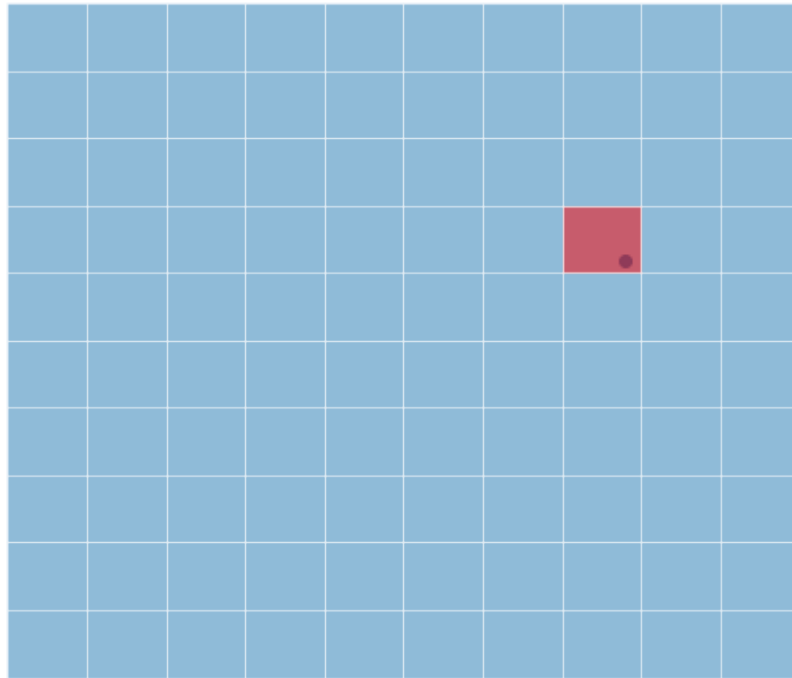
76

In

```
1 # Visualize the point and the nearest grid cell
2 f, ax = plt.subplots(1, 1, figsize=(8, 8))
3
4 # Plot the point of interest
5 gpd.GeoDataFrame([point], columns=['geometry']).plot(ax=ax)
6
7 # Plot the grid
8 gdf_grid.plot(ax=ax, edgecolor='w', alpha=0.5)
9
10 # Highlight the nearest grid cell
11 gdf_grid[gdf_grid.index == nearest].plot(color='red',
12 ax=ax,
13 edgecolor='w',
14 alpha=0.5)
15
```

```
16 ax.axis('off')
```

```
17 plt.show()
```



This example demonstrates how to find the grid cell (or any other geometry) that encloses a specific point using RTree indexing.

54. Introduction to H3 Indexing

[H3](#) is a spatial indexing system developed by designed to partition the world efficiently into hexagonal grids of discrete sizes. Unlike traditional square grids, hexagonal grids have the advantage of reduced distortion and more uniform distance metrics thanks to their symmetry properties. H3 allows for hierarchical indexing, meaning cells at different resolutions nest within each other, providing a scalable solution for spatial analysis.

The H3 system offers various cell [sizes](#) depending on the resolution level, with resolution 0 being the largest (1281 km edge length) and covering a substantial portion of the Earth and higher resolutions providing a more detailed view of the data with hexagons and going down to resolution 15 (around 0.55m). For instance, detailed urban analytics projects usually work using resolutions of 8-11. This versatility makes H3 one of the most popular and widely used spatial indexing systems today.

In the following example, we will define a set of coordinates for San Francisco, CA. Then, using the `h3.geo_to_h3` function we query the H3 index for these coordinates at a specified resolution (in this case, resolution 9). We will also extract the boundary of the H3 cell and find its neighboring cells utilizing the `h3.k_ring` function to find the neighboring H3 cells within a specified distance (k-ring).

In

```
1 # Importing the H3 library
2 import h3
```

3

4 # Define some coordinates for San Francisco, CA

5 latitude, longitude = 37.7749, -122.4194

6

7 # Get the H3 index at a given resolution (e.g., resolution 9)

8 resolution = 9

9 h3_index = h3.geo_to_h3(latitude, longitude, resolution)

10 print("H3 Index:", h3_index, "\n")

11

12 # Get the boundary of the H3 cell

13 h3_boundary = h3.h3_to_geo_boundary(h3_index)

14 print("\nH3 Cell Boundary:")

15 for lat, lon in h3_boundary:

16 print(lat, lon)

17

18 # Get the neighboring H3 cells within a distance of 1 k-ring

19 neighbors = h3.k_ring(h3_index, 1)

20 print("\nNeighboring H3 Cells:")

21 for neighbor in neighbors:

22 print(neighbor)

H3 Index: 89283082803ffff

H3 Cell Boundary:

37.7720104773324 -122.41701147197291

37.773693172998044 -122.4159401398489

37.775197782893386 -122.41719971841657

37.77501967379261 -122.41953062807339

37.77333697800609 -122.42060189084881

37.77183239144092 -122.41934231331864

Neighboring H3 Cells:

8928308280ffff

8928308281bffff

89283082807ffff

8928308280bffff

89283082813ffff

89283082803ffff

89283082817ffff

This example illustrates the basic usage of H3 in obtaining the enclosing hexagon, its boundary coordinates, and neighboring cells of a given location.

55. Visualizing H3 Grids

Once we learn how to obtain the H3 indices shown in the previous section, we can place them on a map. Visualizing H3 grids can be useful for geospatial analysis, as it allows us to see how hexagonal cells overlay on a geographic area. This is particularly beneficial for understanding spatial patterns and performing detailed spatial analysis. In this section, we will demonstrate how to visualize H3 grids by splitting an administrative boundary polygon into hexagonal cells and plotting these cells on a map.

We will start by defining a function `split_poly_to_hexagons` that converts a polygon, such as a city's administrative boundary into a hexagonal grid. The function extracts the coordinates of the boundary polygon and converts it to a GeoJSON format. It then uses the `h3.polyfill` method to generate hexagon-shaped polygons that cover the polygon at the given resolution, which are then packaged into a `GeoDataFrame` using the CRS of the input data.

We then extract Germany's administrative boundaries from the built-in global map data, use to create a hexagonal grid at resolution 4, and visualize the grid overlaid on the German map using Matplotlib.

In

```
1 # Import all necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import Polygon
```



```

4 import matplotlib.pyplot as plt
5 import h3
6
7 # Function to split an administrative boundary into a hexagon grid
8 def split_poly_to_hexagons(admin_gdf, resolution, crs):
9
10     coords = list(admin_gdf.geometry.to_list()[0].exterior.coords)
11     admin_geojson = {"type": "Polygon", "coordinates": [coords]}
12     hexagons = h3.polyfill(admin_geojson,
13 resolution,
14 geo_json_conformant=True)
15
16     hexagon_geometries = {hex_id:
Polygon(h3.h3_to_geo_boundary(hex_id, \
17 geo_json=True)) for hex_id in hexagons}
18     gdf_hexagon_geometries =
gpd.GeoDataFrame(hexagon_geometries.items(),
19 columns=['hex_id', 'geometry'])
20     gdf_hexagon_geometries.crs = crs
21
22     return gdf_hexagon_geometries

```

In

```

1 # Extract the administrative boundary of Germany from the
GeoDataFrame
2 gdf = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
3 gdf_example = gdf[gdf.name == 'Germany']
4

```

5 # Define the H3 resolution

6 resolution = 4

7

8 # Create a hexagonal grid within the boundary of Germany

9 gdf_hexagons = split_poly_to_hexagons(gdf_example, resolution,
gdf_example.crs)

10

11 # Visualize the hexagonal grid

12 f, ax = plt.subplots(1, 1, figsize=(10, 5))

13

14 # Plot the boundary of Germany

15 gdf_example.plot(ax=ax, color='none', edgecolor='k', linewidth=3)

16

17 # Plot the hexagonal grid

18 gdf_hexagons.plot(ax=ax, edgecolor='w', alpha=0.5)

19 ax.axis('off')

20 plt.show()



In this example, we reviewed how to create an H3 hexagon grid from a given input polygon and how to visualize it along the initial input polygon using Matplotlib.

Summary on Spatial Indexing

In this chapter, we reviewed the essential concepts and techniques related to spatial indexing, which are powerful tools for handling large-scale geospatial data. We began by creating a simple spatial index in GeoPandas using its built-in spatial index method, demonstrating how this basic indexing technique can significantly enhance query performance by quickly locating geographic features without scanning the entire dataset. Then, we further explored RTree, a hierarchical spatial index structure that organizes data into a tree of bounding rectangles. We dived into square and hexagon grids, learned about Uber's H3 indexing, and created multiple visualizations aiding the deeper understanding of the spatial indexing concepts we overview.

Geocoding



Geocoding is a widely used concept in spatial technologies and geospatial analysis that involves converting addresses into geographic coordinates and vice versa. This capability is essential for a wide range of applications, from mapping and spatial analysis to location-based services and data validation. While there are many proprietary geocoding services available, in this chapter, we will focus on a free solution using the GeoPy library, which interfaces with the Nominatim geocoder based on OpenStreetMap data.

We will start by demonstrating how to geocode a single address as well as how to do batch geocoding. Then, we explore reverse geocoding, which transforms geographic coordinates back into human-readable addresses. Then, we link these geocoding techniques to the data manipulation framework of Pandas and GeoPandas. By the end of this chapter, we will have a comprehensive understanding and hands-on experience of geocoding in Python.

56. Geocoding a Single Address Using GeoPy

Geocoding is the process of converting addresses into geographic coordinates, which can then be used to place markers on a map, perform spatial analysis, or simply understand the location in terms of latitude and longitude. [GeoPy](#) is a popular Python library that simplifies the geocoding process by providing a consistent interface. We will further explore it below.

Here, we will demonstrate how to geocode a single address using GeoPy. We will use the Nominatim geocoder class from GeoPy, which is based on [OpenStreetMap](#) data, to convert individual example addresses into geographic coordinates. First, we need to create a geolocator object with a user agent. The user agent is a required parameter to identify our application to the geocoding service. We then use the geocode method of the geolocator object to convert the example address into geographic coordinates. The method returns a location object containing various information, some of which we print out to the code cell's output.

In

```
1 # Import the Nominatim geocoder from GeoPy
2 from geopy.geocoders import Nominatim
3
4 # Create a geolocator object with a user agent
5 geolocator =
6
7 # Geocode an address in New York City
```

```
8 location_nyc = Building, \  
9     175, 5th Avenue, Manhattan, New York City")  
10 print("New York City Address:")
```

```
11  
12 # Print the full address  
13  
14  
15 # Print the latitude and longitude  
16
```

New York City Address:

Flatiron Building, 175, 5th Avenue, Manhattan Community Board 5,
Manhattan, New York County, City of New York, New York, 10010,
United States
(40.741059199999995, -73.98964162240998)

In

```
1 # Geocode an address in Budapest  
2 location_budapest = Vadasz utca 35b")  
3 print("\nBudapest Address:")  
4  
5 # Print the full address  
6  
7  
8 # Print the latitude and longitude  
9
```

Budapest Address:

35b, Vadász utca, Lipótváros, V. kerület, Budapest, Közép-Magyarország,
1054, Magyarország
(47.5068163, 19.0535141)

The cell outputs show the exact address names and their longitude-latitude coordinate pairs for both the New York City (Flatiron Building, 175, 5th Avenue, Manhattan, New York City) and the Budapest (Budapest, Vadasz utca 35b) examples.

57. Reverse Geocoding Coordinates

Reverse geocoding is the process of converting geographic coordinates into a human-readable address. This is particularly useful for understanding the location details of a given point on a map, enabling applications such as displaying location information and human-readable addresses for user clicks on a map, validating geospatial data, or enriching datasets with address information. GeoPy provides a straightforward interface for reverse geocoding, making it easy to convert coordinates into addresses.

In this section, we will demonstrate how to reverse geocode coordinates using GeoPy. We will use the Nominatim geocoder again and reverse engineer the exact addresses of the two example coordinates that we geocoded in the previous section, one in New York City and another in Budapest.

In

```
1 # Import the Nominatim geocoder from GeoPy
2 from geopy.geocoders import Nominatim
3
4 # Create a geolocator object with a user agent
5 geolocator =
6
7 # Reverse geocode coordinates in New York City
8 location_nyc = -73.98964162240998")
9 print("Address for NYC Coordinates:")
```

```
10 # Print the full address
11
12 # Reverse geocode coordinates in Budapest

13 location_budapest = 19.0535141")
14 print("\nAddress for Budapest Coordinates:")
15 # Print the full address
```

Address for NYC Coordinates:

Flatiron District, 5th Avenue, Manhattan Community Board 5, Manhattan,
New York County, City of New York, New York, 10010, United States

Address for Budapest Coordinates:

35b, Vadász utca, Lipótváros, V. kerület, Budapest, Közép-Magyarország,
1054, Magyarország

When comparing these addresses with the previous section, we can confirm that the reverse geocoding was indeed successful.

58. Batch Geocoding Multiple Addresses

Geocoding multiple addresses efficiently is essential in many geospatial applications, such as processing and preparing large datasets, creating maps, or performing spatial analyses. Batch geocoding allows us to convert a list of addresses into geographic coordinates in a single process, rather than geocoding each address individually. GeoPy supports this by providing tools to handle rate limiting and batch processing seamlessly.

In this section, we will demonstrate how to batch geocode multiple addresses stored in a Python list using GeoPy. We will again use the Nominatim geocoder, which is based on the free OpenStreetMap data, and employ the RateLimiter class to manage the rate of requests, ensuring we do not exceed the usage limits of the geocoding service.

In

```
1 # Import the Nominatim and the RateLimiter from GeoPy
2 from geopy.geocoders import Nominatim
3 from geopy.extra.rate_limiter import RateLimiter
4
5 # Create a geolocator object with a user agent
6 geolocator =
7
8 # List of addresses to be geocoded
9 addresses = ["Flatiron Building, 175, 5th Avenue, Manhattan, New York
City",
10             "Budapest, Vadasz utca 35b"]
```

```
11
12 # Create a RateLimiter to manage the rate of requests

13 geocode =
14
15 # Geocode each address in the list
16 locations = [geocode(address) for address in addresses]
17
18 # Print the geocoded locations
19 for loc in locations:
20
```

Flatiron Building, 175, 5th Avenue, Manhattan Community Board 5,
Manhattan, New York County, City of New York, New York, 10010,
United States (40.741059199999995, -73.98964162240998)
35b, Vadász utca, Lipótváros, V. kerület, Budapest, Közép-Magyarország,
1054, Magyarország (47.5068163, 19.0535141)

As the cell's output shows, we geocoded each address and outputted the stored location name and coordinate within for loops as well.

59. Handling Missing Geocodes

When working with large-sized real-world data, it is common to encounter at least a few addresses that cannot be geocoded for various reasons, such as inaccuracies, typos, or the address not being present in the geocoding service's database. Handling such missing geocodes without breaking our batch geocoding algorithms is essential to ensure that our geospatial processes stay intact when encountering such data clarity issues.

In this section, we will demonstrate how to handle missing geocodes using GeoPy. We will geocode a list of addresses, some of which may not return valid coordinates. We will check if each address is successfully geocoded and handle the cases where geocoding fails by providing a suitable message.

In

```
1 # Import the Nominatim and the RateLimiter from GeoPyfrom
2 from geopy.extra.rate_limiter import RateLimiter
3
4 # Create a geolocator object with a user agent
5 geolocator =
6
7 # List of addresses, including an unknown place
8 addresses = ["185 5th Avenue NYC", "175 5th Avenue NYC",
9 "UnknownPlace"]
10 # Create a RateLimiter to manage the rate of requests
```

```
11 geocode =
12
13 # Geocode each address in the list
14 locations = [geocode(address) for address in addresses]
15
16 # Print the geocoded locations, handling missing geocodes
17 for idx, loc in enumerate(locations):
18     if loc:
19         # Print the full address if found
20         print(addresses[idx] + ": FOUND - " ,
21
22         # Print the address and message if the address cannot be
geocoded
23         print(addresses[idx] + ": NOT FOUND")
24     print()
```

185 5th Avenue NYC: FOUND - 5th Ave. Market, 185, 5th Avenue,
Brooklyn, Kings County, City of New York, New York, 11217, United
States

175 5th Avenue NYC: FOUND - Flatiron Building, 175, 5th Avenue,
Manhattan Community Board 5, Manhattan, New York County, City of
New York, New York, 10010, United States

UnknownPlace: NOT FOUND

As a result of the loop shows, we were able to geocode the two known
addresses, however, the third place we called UnknownPlace was truly

impossible to geocode.

60. From Geocoding to GeoDataFrame

In geospatial analysis, we often need to convert a batch of addresses into a GeoDataFrame to ensure compatibility with already existing data tables. Hence, we now learn how to quickly turn the results of our batch geocoding process into a GeoDataFrame.

We will use GeoPy to geocode a list of addresses, extract the latitude and longitude, and then package these results into a GeoDataFrame by creating points geometries using Shapely.

In

```
1 # Importing all the necessary libraries
2 from geopy.geocoders import Nominatim
3 from geopy.extra.rate_limiter import RateLimiter
4 from shapely.geometry import Point
5 import geopandas as gpd
6
7 # Create a geolocator object with a user agent
8 geolocator =
9
10 # List of addresses to be geocoded
11 addresses = ["175 5th Avenue NYC", "Budapest, Vadasz utca 15"]
12
13 # Create a RateLimiter to manage the rate of requests
14 geocode =
15
```



```
16 # Geocode each address in the list and store the results
17 geo_data = []

18 locations = [geocode(address) for address in addresses]
19 for loc in locations:
20     if loc:
21
22         'geometry':
23
24 # Convert the geocoding results into a GeoDataFrame
25 gdf_geo = 'geometry'])
26
27 # Display the GeoDataFrame
28 gdf_geo
```

Out



Packaging geocoding results into a GeoDataFrame allows us to store and use geocoded addresses and their coordinates in a GeoDataFrame and prepare the data to later perform various spatial operations, visualize the data on maps, and integrate it with other geospatial datasets.

61. Geocoding within a DataFrame

Geocoding addresses within a Pandas DataFrame allows us to seamlessly integrate geographic coordinates with our existing tabular data. This process involves applying a geocoding function to each row in the DataFrame to obtain latitude and longitude for each address. By doing this, we can enrich our dataset with geospatial information without the need for manual data manipulation.

In this section, we will define a function to geocode addresses and then apply this function to a column in the DataFrame, creating new columns for latitude and longitude. At the end, we display the resulting DataFrame. By integrating the previously shown methods for creating GeoDataFrames, then, we can also quickly turn this Pandas DataFrame into a GeoDataFrame for further analysis.

In

```
1 # Necessary imports
2 import pandas as pd
3 from geopy.geocoders import Nominatim
4
5 # Create a geolocator object with a user agent
6 geolocator =
7
8 # Sample DataFrame with city names
9 gdf_cities =
10     'name': ['New York', 'Budapest', 'Paris', 'London', 'Berlin']
```

```
11 })
12
13 # Function to geocode an address
14 def geocode_address(address):
15     location =
16     return
17         'Longitude':
18         if location else
19         'Longitude':
20
21 # Apply the geocoding function to the 'name' column to
22 # create 'Latitude' and 'Longitude' columns
23 gdf_cities[['Latitude', 'Longitude']] =
24
25
26 # Display the updated DataFrame
27 gdf_cities
```

Out



Summary on Geocoding

In this chapter, we explored various aspects of geocoding techniques, turning addresses into coordinates, using the GeoPy library, and leveraging the free Nominatim geocoder based on OpenStreetMap data. We began by demonstrating how to geocode a single address, converting them into geographic coordinates. Following this, we reverse geocoded coordinates to obtain human-readable addresses, showcasing the bidirectional capabilities of geocoding.

We then addressed batch geocoding, which is essential for processing multiple addresses efficiently, and demonstrated how to handle missing geocodes to ensure robust data processing. Furthermore, we turned geocoding results into a GeoDataFrame, allowing for advanced geospatial analysis using GeoPandas. Lastly, we showed how to integrate geocoding within a Pandas DataFrame, enriching existing tabular data with geographic coordinates.

Raster Data in Python



Earlier, we reviewed that there are two main types of data formats used in geospatial analytics: vector data and raster data. While we have been extensively studying vector data so far, it is now time to learn about raster data.

Raster data is a core component of geospatial analysis, consisting of a grid of pixels where each pixel holds one or more data points or bands, representing information such as population density, land cover, or elevation. In the case of multi-band data, each band in a pixel can store different types of data, such as spectral information from various wavelengths in remote sensing. This type of data is typically stored in file formats such as PNG or GeoTIFF and can come in multiple resolutions, each offering different levels of detail and types

of data layers. Efficient handling and processing of raster data are crucial for applications in remote sensing, environmental modeling, and many more.

In this chapter, we will explore techniques and tools in Python to handle large raster datasets. We will start by demonstrating how to read and write raster data, followed by methods to clip rasters to focus on specific regions. We recap the topic of map projections and explore a few visualization techniques. We will learn about more advanced analytical operations as well, such as computing zonal statistics, converting raster grids into vector data, and downsampling large raster files.

Global Population Raster Data

Throughout this notebook, we will rely on various versions of the [GHSL - Global Human Settlement Layer](#) raster database, which contains global population estimates in raster data format.

This dataset is freely available and was created by the European Union's Copernicus Team at the Joint Research Center. According to their website, their aim in creating this data set was to create new global spatial information and publicly available knowledge describing human presence on Earth. This dataset operates under a fully open and free data license policy. We can further study the detailed technical documentation

For the following explorative sections, we may download any raster file we wish from their website, where we can select a prepared transformed version of the GHSL data set from a variety of temporal epochs, two coordinate systems, and a handful of spatial resolutions. For instance, from the GHS-POP product, we can select the epoch of 2030 (future projection) with a resolution of 100m in the Mollweide coordinate system.

Product



Epoch



Resolution



GHS population grid (R2023)

[Read the technical details for this product](#)

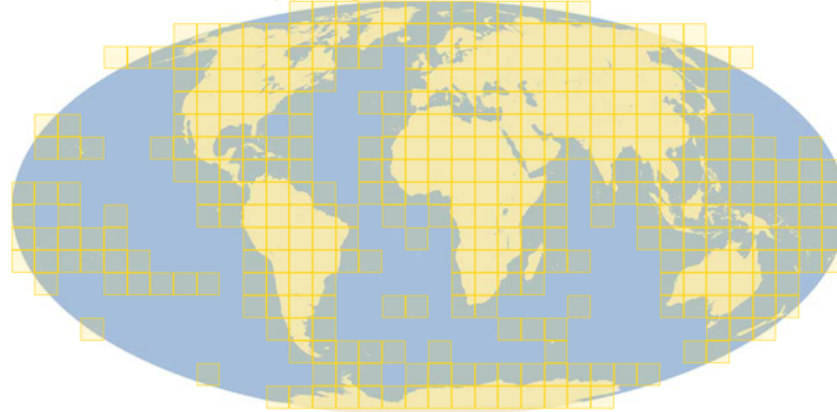
Current selection:

Product: **GHS-POP**, epoch: **2030**, resolution: **100m**, coordinate system: **Mollweide**

① To be noted that some variation might be available only for a certain product. The options not available for a product are disabled and greyed out.

Download by tiles (click on each box to download a single tile):

[Interactive visualisation of the GHS population grid \(R2023\)](#)



[Download the shapefile containing the tiling schema used in the map above.](#)

[Hover over a tile to see its boundaries](#)

Global download (get the chosen data in a single file):

[Download the global GHS_POP_E2030_GLOBE_R2023A_54009_100_V1_0 dataset in a single file](#) or click on a tile on the map above to download the corresponding tile.

The download links are updated reflecting the selection of the filters on the left of the map.

62. Reading Raster Data

Raster data is a fundamental type of geospatial data used in various applications, including remote sensing and environmental modeling. In this section, we will learn how to read raster data using Python. As an example, we will use the Global Human Settlement Layer (GHSL) data for the year 2030, using the WGS84 coordinate reference system (CRS) and a resolution of 30 arcseconds, stored in a GeoTIFF format. We note that the resolution is defined in grid cells with a size of 30 arcseconds, referring to the fact that the WGS84 (EPSG:4326) coordinate system stores locational information in longitude and latitude angles.

In the example below, first, we import the [rasterio](#) library, which is the most common package for handling raster data in Python. Then, we define the path to the raster file based on the downloaded file's name and location. Using we open this file and read the data using the `read()` method. Here, we can specify which band we would like to read - which, in the case of the population data, is not optional as we only have one band containing the population data. However, for instance, in the case of satellite images, the input data usually provides multiple bands, such as red, green, and blue channels.

Then, we also extract additional information, such as the raster profile, bounds, resolution, and statistical values (mean, minimum, and maximum) from the raster data. The profile contains metadata about the raster file, such as the data type, number of bands, and CRS, which we print out for further observation.

In

```
1 # Import rasterio
2 import rasterio

3
4 # Define the raster file path
5 folder = 'GHS_POP_E2030_GLOBE_R2023A_4326_30ss_V1_0'
6 file_name =
'GHS_POP_E2030_GLOBE_R2023A_4326_30ss_V1_0.tif'
7 raster_file = folder + '/' + file_name
8
9 # Read the raster file
10 with as src:
11     # Read the raster data
12     raster_data =
13
14     # Read additional information
15     profile =
16     bounds =
17     resolution =
18     mean_val =
19     min_val =
20     max_val =
21
22 # Print the extracted information
23 print("Profile:", profile)
24 print("Bounds:", bounds)
25 print("Resolution:", resolution)
26 print("Mean value:", mean_val)
27 print("Minimum value:", min_val)
28 print("Maximum value:", max_val)
```

```
Profile: {'driver': 'GTiff', 'dtype': 'float64', 'nodata': None, 'width': 43202,
'height': 21384, 'count': 1, 'crs': CRS.from_epsg(4326), 'transform':
Affine(0.008333333300326923, 0.0, -180.00791593130032,
      0.0, -0.00833333329979504, 89.0995831776456), 'blockxsize': 256,
'blockysize': 256, 'tiled': True, 'compress': 'lzw', 'interleave': 'band'}
Bounds: BoundingBox(left=-180.00791593130032,
bottom=-89.10041610517152, right=180.00874930942342,
top=89.0995831776456)
Resolution: (0.008333333300326923, 0.00833333329979504)
Mean value: 9.250757067224743
Minimum value: 0.0
Maximum value: 394367.2599414062
```

Reading and analyzing raster data is a fundamental step in geospatial analysis. In this example, we used the GHSL data to read and extract population estimates for the year 2030 in WGS84 CRS.

As the code block's output shows, this raster data, data stored in a GeoTIFF format, has a data type of float64. It has a width of 43,202 pixels and a height of 21,384 pixels, containing a single band of data. The file is tiled with block sizes of 256x256 pixels and compressed using LZW compression.

The spatial extent of the data covers the entire globe, with bounds ranging from -180.0079 to 180.0087 degrees in longitude and -89.1004 to 89.0996 degrees in latitude. The resolution of each pixel is approximately 0.0083

degrees. The mean population value within the raster is approximately 9.25, with a minimum value of 0 and a maximum value of 394,367.26, indicating significant variability in population density across the globe. Here, we also note that the population values, indicating the headcount of inhabitants, are modeled values and come from a series of data processing steps, from data integration and enhancement to interpolation. Hence, the maximum (and all other) values stored within the grid are float, not integer, as we would expect.

63. Clip the Raster Data File Using GeoPandas

Clipping a raster data file to a specific region of interest is a common task in geospatial analysis. This process allows us to focus on a particular area, reducing the dataset size and improving processing efficiency.

In this section, we will demonstrate how to clip the GHSL raster data file using GeoPandas and By using this tool, we will narrow down the global population raster map to a subset that only focuses on Europe. We will first read the raster file, as demonstrated in the previous section. Then, we will use from Shapely to capture the box that covers most of Europe. Finally, we use this Shapely object to clip the raster. As the of the raster file shows, the metadata of the clipped raster is updated accordingly to reflect the new dimensions of the smaller, clipped raster data set.

In

```
1 # Import all the libraries we use
2 import rasterio
3 from rasterio.mask import mask
4 from shapely.geometry import box
5 import geopandas as gpd
6
7 # Define the bounding box for Europe
8 bbox = 34.0, 40.0, 72.0)
9 gdf = bbox},
10
11 # Open the raster file
```

```
12 with as src:
13     # Clip the raster using the bounding box
14     out_image, out_transform = mask(src,
15     out_meta =
16
17     "driver": "GTiff",
18     "height":
19     "width":
20     "transform": out_transform
21 })
22
23 # Print the profile of the clipped raster
24 print("Profile of the clipped raster:")
25 for key, value in
26     print(f'{key}: {value}')
```

Profile of the clipped raster:

driver: GTiff

dtype: float64

nodata: None

width: 6001

height: 4561

count: 1

crs: EPSG:4326

transform: | 0.01, 0.00,-10.01|

| 0.00,-0.01, 72.01|

| 0.00, 0.00, 1.00|

The output shows that the clipped raster file, similar to the original, is saved in GeoTIFF format with a pixel data type of However, the raster dimensions are 6,001 pixels in width and 4,561 pixels in height, indicating the reduced area of the clipped region while retaining a single band and the original WGS84 coordinate reference system.

On the output, we see another concept that we should lay out and learn for raster data analysis - it is the transform attribute. In geospatial data, a transform is a mathematical function that converts pixel coordinates in a raster dataset to spatial coordinates, allowing each pixel to map to a specific location on Earth's surface. In the transform is represented as a so-called [affine transformation](#) which includes translation, scaling, rotation, and shearing. The matrix $\begin{bmatrix} 0.01 & 0.00 & -10.01 \\ 0.00 & -0.01 & 72.01 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}$ indicates that each pixel is 0.01 map units wide and tall (with the y-coordinate increasing downward), the x-coordinate of the top-left pixel is -10.01, and the y-coordinate is 72.01. This matrix helps convert the raster's pixel coordinates to geographic coordinates, ensuring accurate spatial representation. While the theoretical parts can be tricky, in rasterio, we have everything at hand to use this transform to efficiently convert between pixel and spatial coordinates with ease.

64. Writing Raster Data

Once we have clipped our raster data to a region of interest, the next step is often to save this modified data to a new file for further use or analysis. Writing raster data involves specifying the output file path and using Rasterio to write the clipped raster data for a new file while preserving its metadata.

In the example below, first, we read and clip the original raster file to the bounding box of Europe. Then, we demonstrate how to write the clipped raster data into a new GeoTIFF file. This involves defining the output file path and using the open function from Rasterio in write mode to save the raster data.

In

```
1 # Import all the libraries we use
2 import rasterio
3 from rasterio.mask import mask
4 from shapely.geometry import box
5 import geopandas as gpd
6
7 # Define the bounding box for Europe
8 bbox = 34.0, 40.0, 72.0)
9 gdf = bbox},
10
11 # Open the raster file
12 with as src:
```



```
13 # Clip the raster using the bounding box
14 out_image, out_transform = mask(src,
15 out_meta =

16
17     "driver": "GTiff",
18     "height":
19     "width":
20     "transform": out_transform
21 })
22
23
24 # Define the output file path
25 clipped_raster_file = 'clipped_raster_europe.tif'
26
27 # Save the clipped raster
28 with 'w', as dest:
29
30
31 print(f'Clipped raster saved to {clipped_raster_file}')
```

Clipped raster saved to clipped_raster_europe.tif

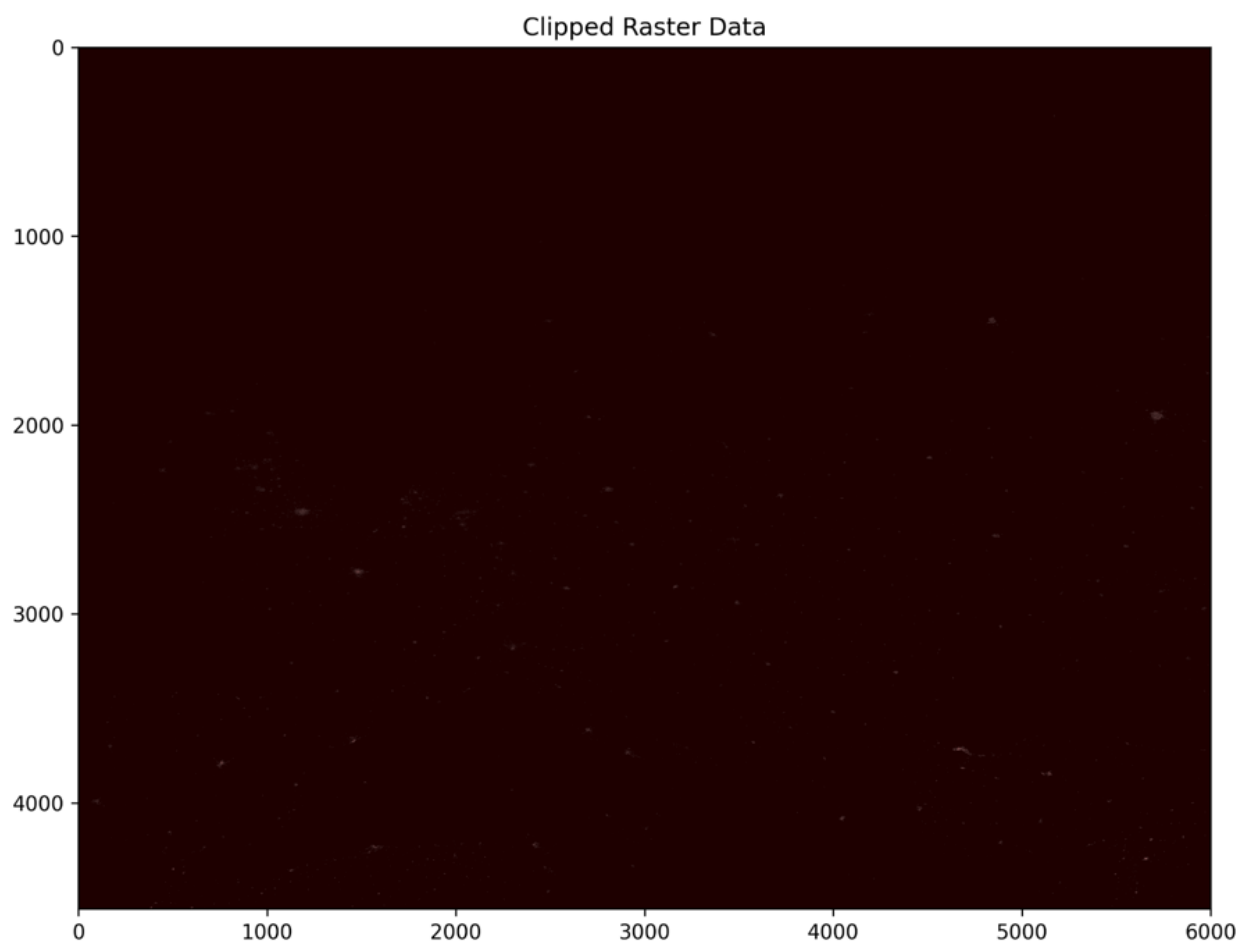
65. Visualizing Raster Data

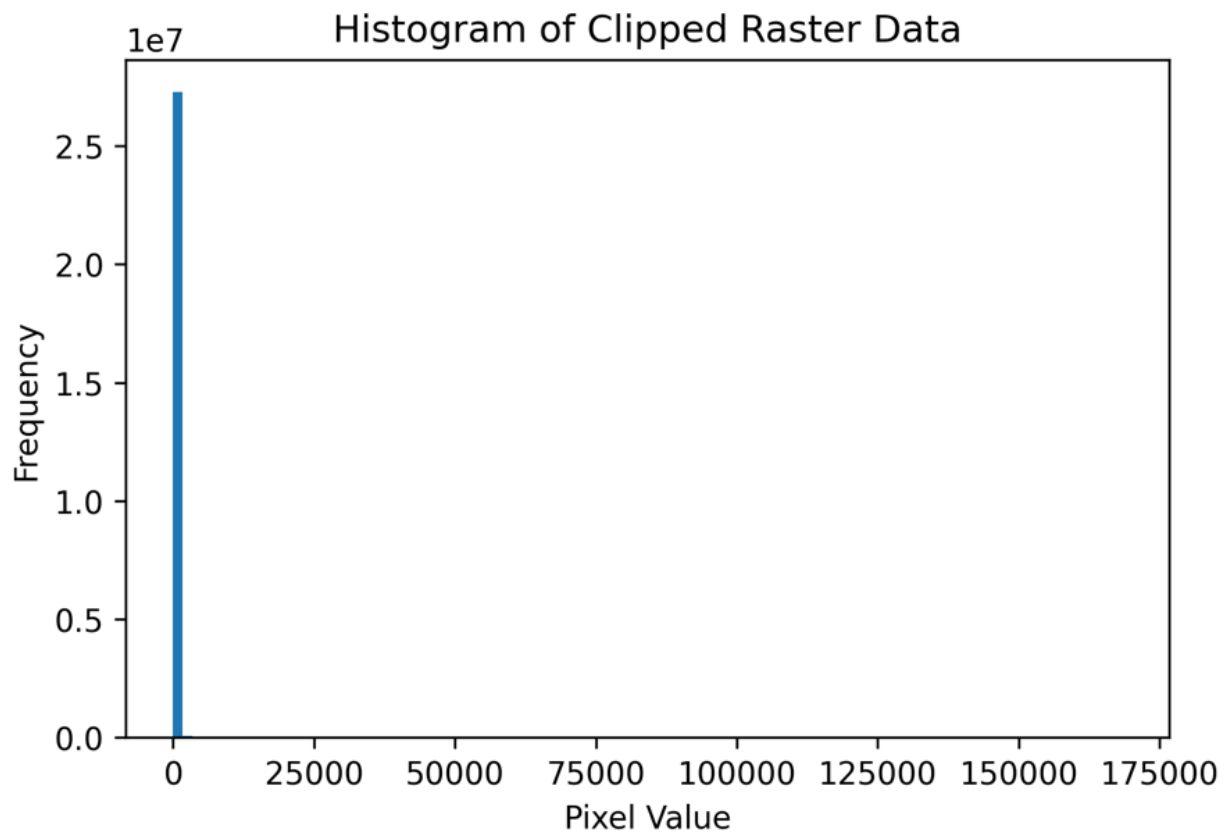
Visualizing raster data is an important step in geospatial analysis as it helps to understand the distribution and patterns within the data. In this code example, we first import the necessary libraries including `show` from `rasterio` and `plt` from `matplotlib.pyplot`. We then open the clipped raster file using `rasterio.open` and read the data with `rasterio.read`. The `show` function from `Rasterio` is used to visualize the raster data with a specified color map and `matplotlib.pyplot` is used to set the title of the plot. Additionally, we create a histogram of the raster data values using `plt.hist` to observe the pixel-level population value distribution and interpret the visualized raster map.

In

```
1 # Import necessary libraries
2 import rasterio
3 from rasterio.mask import mask
4 from rasterio.plot import show
5 import matplotlib.pyplot as plt
6
7 # Define the output file path
8 clipped_raster_file = 'clipped_raster_europe.tif'
9
10 # Open the clipped raster file
11 with rasterio.open(clipped_raster_file) as src:
12     out_image, out_meta = mask(src, [0], crop=True)
13     out_image = out_image[0]
14
15 # Visualize the clipped raster
```

```
16 fig, ax = 10))  
  
17 show(out_image,  
18 Raster Data')  
19  
20  
21 # Visualize the histogram of the raster data  
22  
23 Value')  
24  
25 of Clipped Raster Data')  
26
```





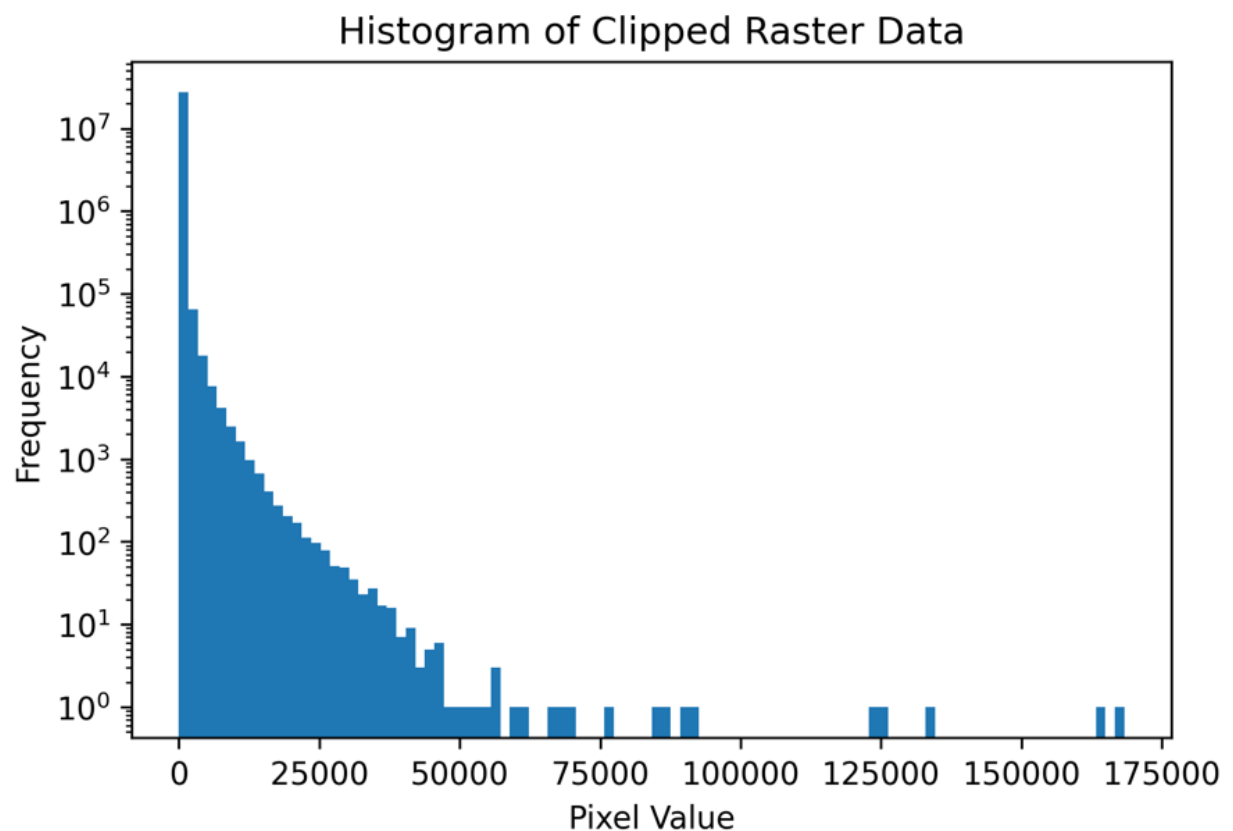
While all the visualization parameters seem to be properly set, we still only see a mostly black canvas, where we were expecting to see a population map of Europe instead. The reason behind this lies in the histogram, as there are just a few outlier values with extremely large population levels, which distort the coloring of the raster map so much that nothing will be visible. However, we can quickly overcome this issue and get a better view of Europe's population landscape by plotting the histogram with the y-scale set to logarithmic with and then apply this to the 2D raster map as well.

In

```
1 # Visualize the histogram of the raster data
2
3
4 Value')
5
```

6 of Clipped Raster Data')

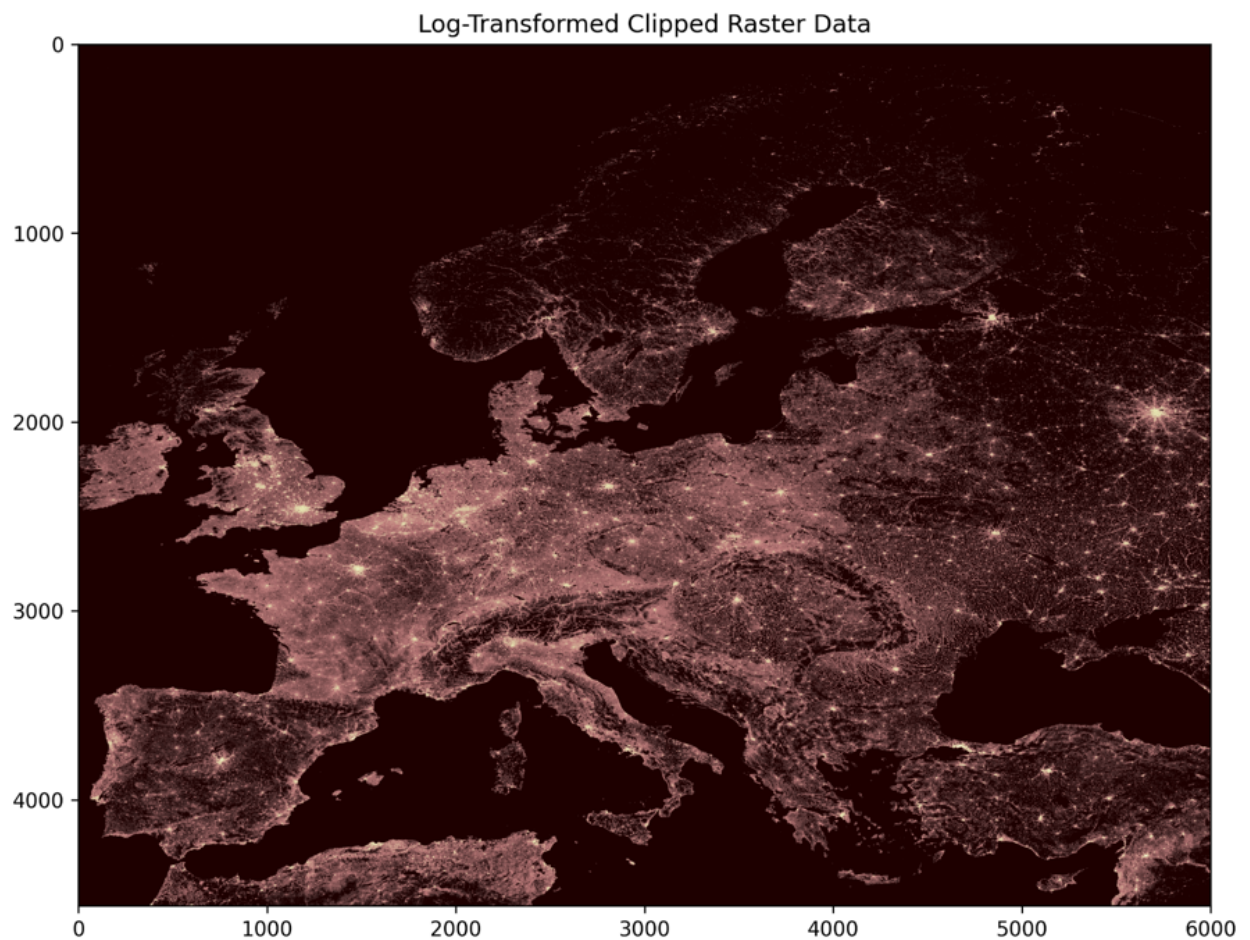
7



In

```
1 # Import numpy
2 import numpy as np
3
4 # Apply a logarithmic transformation to the data
5 log_image =
6
7 # Visualize the transformed raster
8 fig, ax = 10))
```

```
9 show(log_image,  
10 Clipped Raster Data')  
11
```



After doing the logarithmic scaling, we already have a familiar image - Europe with all the populated areas highlighted.

66. Histogram Equalization on Raster Data

Logarithmic scaling is a good basic solution for visualizing skewed raster data, as it helps to stretch out the low values and compress the high values, making the data distribution more visible.

However, depending on the input data, sometimes the so-called histogram equalization can be a more effective method for enhancing the contrast in the data. Histogram equalization works by redistributing the pixel values so that they span the entire range of possible values more evenly. This is achieved by calculating the cumulative distribution function (CDF) of the pixel values and then using this CDF to remap the original pixel values. As a result of this process, pixel values that are more frequent in the original image will be spread out over a larger range in the equalized image, enhancing the contrast. The result, usually, is an improved overall contrast of the image.

Let's see it with the previously clipped European population raster grid!

In

```
1 # Import necessary libraries
2 import rasterio
3 from rasterio.plot import show
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from sklearn.preprocessing import QuantileTransformer
7
```

```
8 # Define the output file path
9 clipped_raster_file = 'clipped_raster_europe.tif'
10
```

```
11 # Open the clipped raster file
12 with as src:
13     out_image =
14     out_transform =
15     out_meta =
```

In

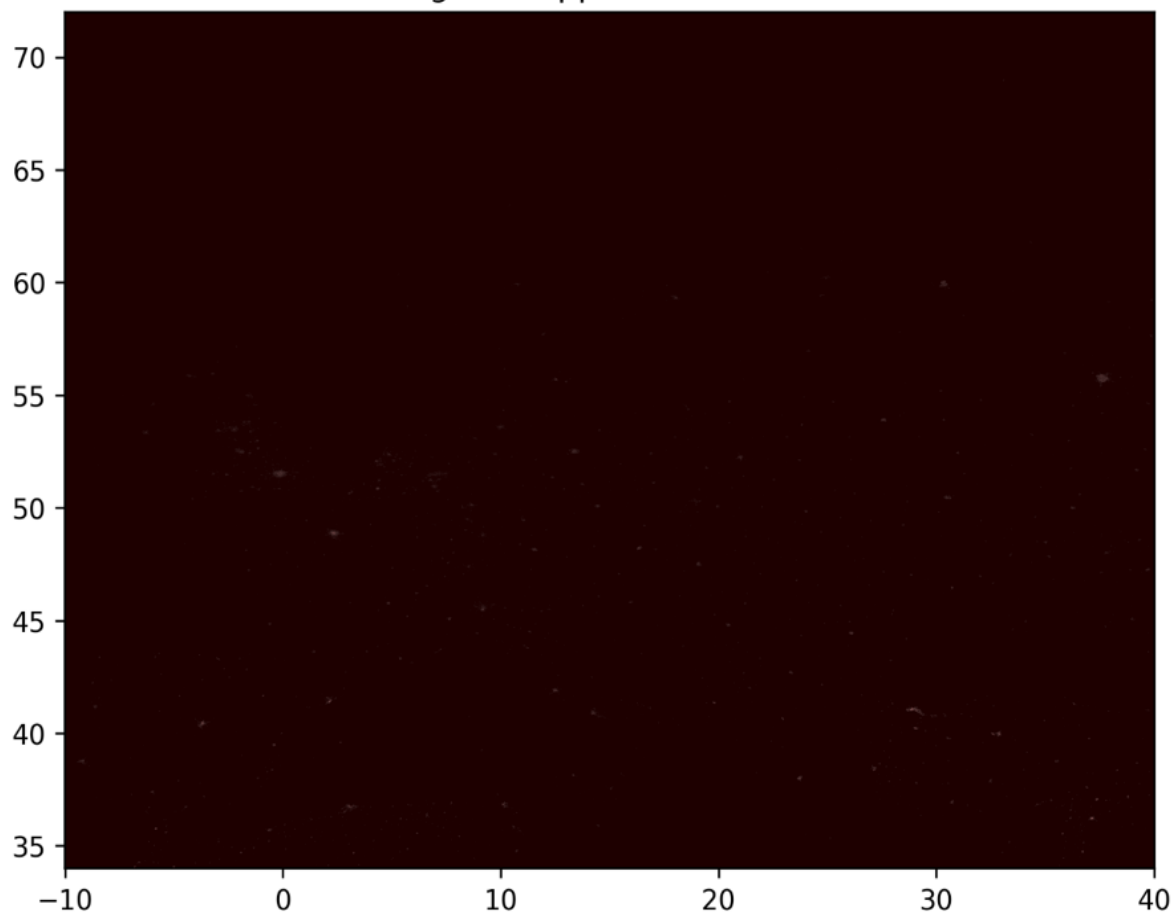
```
1 # Apply histogram equalization for better contrast
2 def histogram_equalization(image):
3     # Flatten the image array and calculate the histogram
4     image_flat =
5     histogram, bin_edges =
6
7     # Cumulative distribution function
8     cdf =
9     cdf_normalized = cdf * /
10
11     # Use linear interpolation of the cdf to find new pixel values
12     image_equalized = cdf_normalized)
13
14     return
15
16 # Apply the equalization
17 equalized_data = histogram_equalization(out_image)
```

In

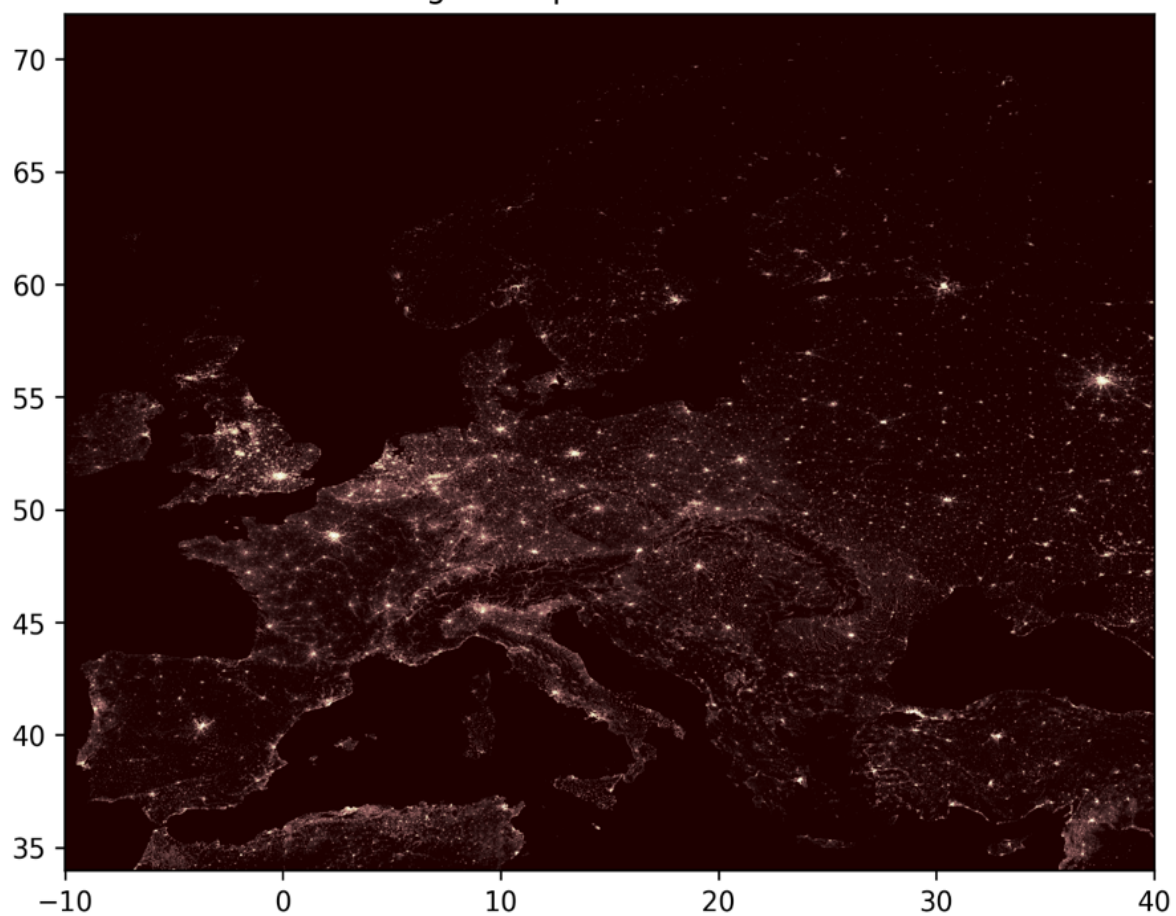
```
1 # Visualize the clipped and equalized raster

2 fig, ax = 1, 10))
3
4 # Original visualization
5 show(out_image,
6 Clipped Raster Data')
7
8 # Enhanced visualization with histogram equalization
9 show(equalized_data,
10 Equalized Raster Data')
11
12 # Final touches on the plot
13
14
```

Original Clipped Raster Data



Histogram Equalized Raster Data



While in this section, we used a more advanced histogram equalization technique to visualize the widely spread population data; it is hard to tell whether this method or the much simpler logarithmic scaling method provided a better solution in the GHSL data. Nevertheless, depending on the input data, using one or the other method may provide significantly better results, so they are both handy tools in the geospatial data analysis stack.

67. Applying Simple Functions on Raster Data

In this section, we will explore how to apply simple functions to raster data. As an example, we will define a set of reclassification rules to categorize the data into different classes based on specific thresholds. Such operations are fundamental in geospatial analysis and can help transform raw data into more meaningful insights.

In the code below, we start by opening the clipped raster file and reading its data using the rasterio library. Then, we write a simple function called which contains reclassification rules to categorize the continuous population data values into three classes: low, medium, and high population. Finally, we use the `numpy.vectorize` function to apply these rules to the entire raster dataset and visualize the results.

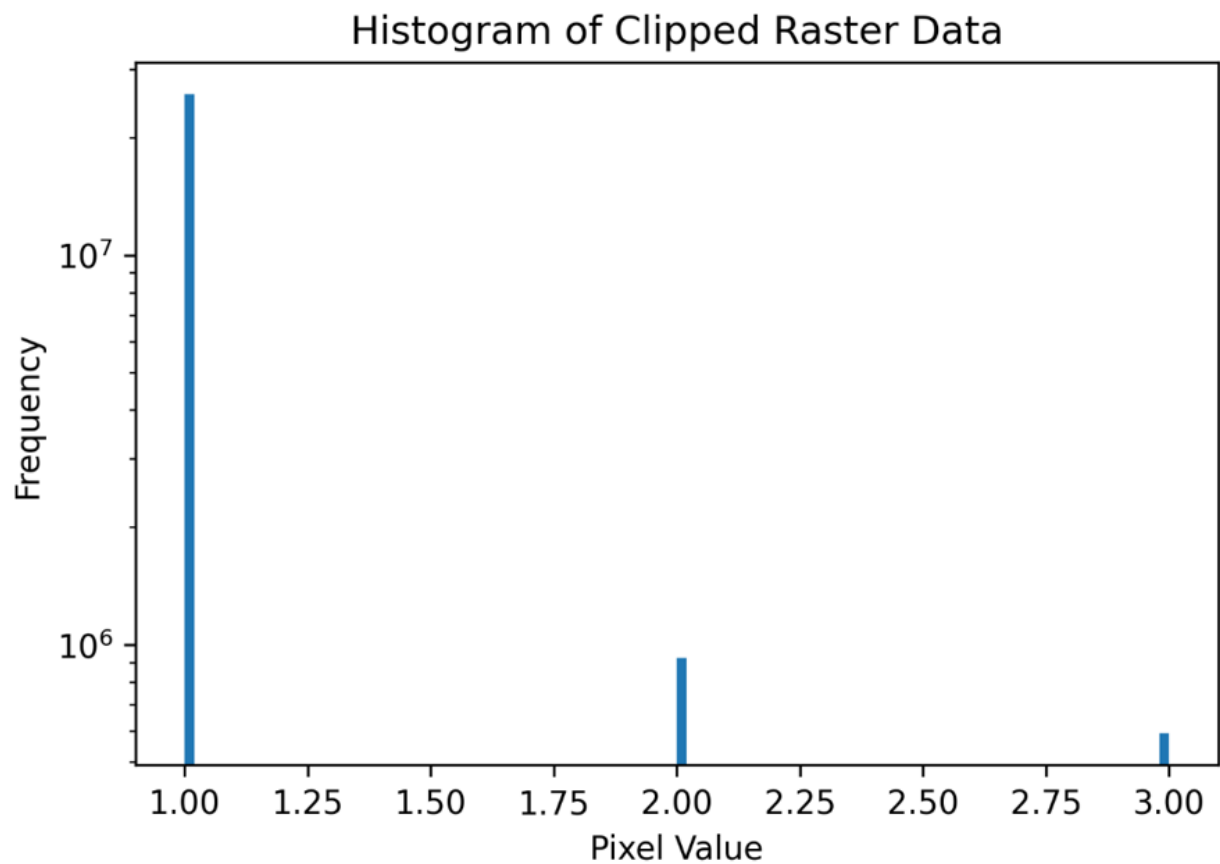
In

```
1 # Import necessary libraries
2 import rasterio
3 import numpy as np
4 from rasterio.plot import show
5
6 # Define the output file path
7 clipped_raster_file = 'clipped_raster_europe.tif'
8
9 # Open the raster file
10 with as src:
11     raster_data =
12     profile =
13
```

```
14 # Define reclassification rules

15 def reclassify(value):
16     if value < 50:
17         return 1 # Low population density
18     elif value < 200:
19         return 2 # Medium population density
20
21     return 3 # High population density
22
23 # Apply reclassification
24 reclassified =
25
26 # Visualize the reclassified raster
27 show(reclassified,
28
29 # Visualize the histogram of the reclassified raster data
30
31
32 Value')
33
34 of Clipped Raster Data')
35
```





In this section, we demonstrated how to apply simple functions to raster data using rasterio and numpy to transform the original population raster grid into a categorical map differentiating low, medium, and high population levels. Then, we visualized this categorical raster map and plotted the histogram of its values to show the frequencies of the three categories.

68. Reprojecting Raster Data

We have already explored the topic of reprojecting geospatial vector data inside out, and now we will extend this to raster data. As a reminder, geospatial data comes in different coordinate reference systems (CRS), which encode mathematical rules to map the three-dimensional earth to the two-dimensional plane. Different CRSs serve different purposes, while converting between them - reprojecting the data - is essential when combining datasets with different CRSs or when aligning data to specific tasks.

In this section, we will demonstrate how to reproject a raster file using the rasterio library in Python. We will again start by opening the clipped Europe population raster file and reading its data. Then, we will define the target CRS (in this case, EPSG:3857 or Web Mercator) and calculate the transform (remember, the raster - spatial transformation matrix) and dimensions for the output raster. Using the `calculate_default_transform` and `reproject` functions from we will transform the raster data to the target CRS, save the reprojected raster to a new file, and print the results to validate our process.

In

```
1 # Import necessary libraries
2 import rasterio
3 from rasterio.warp import calculate_default_transform, reproject
4 from rasterio.enums import Resampling
5
```

```
6 # Path to your input raster file
7 raster_file = 'clipped_raster_europe.tif'

8
9 # Define the target CRS (EPSG:3857 for Web Mercator)
10 target_crs = 'EPSG:3857'
11
12 # Open the raster file
13 with as src:
14     # Read the raster data
15     raster_data =
16     original_transform =
17     original_crs =
18
19     # Calculate the transform and dimensions of the output raster
20     transform, width, height = calculate_default_transform(
21         target_crs,
22
23     # Update the profile for the output raster
24     profile =
25
26     'crs': target_crs,
27     'transform': transform,
28     'width': width,
29     'height': height
30 })
31
32 # Reproject the raster data

33 reprojected_raster = width),
34 reproject(
35
```

```
36
37
38
39
40
41
42
43 print(f'Reprojected raster transformation completed.")
44
45 # Compare their CRS of the original and the reprojected raster data
46 print(f'Original CRS: {original_crs}")
47 print(f'Original Transform: {original_transform}")
48
49 print(f'New CRS: {profile['crs']}")
50 print(f'New Transform: {transform}")
```

Reprojected raster transformation completed.

Original CRS: EPSG:4326

Original Transform: | 0.01, 0.00,-10.01|

| 0.00,-0.01, 72.01|

| 0.00, 0.00, 1.00|

New CRS: EPSG:3857

New Transform: | 1263.50, 0.00,-1114076.18|

| 0.00,-1263.50, 11756037.07|

| 0.00, 0.00, 1.00|

As the output says, the reprojection of the raster data was successfully completed. The original raster used the EPSG:4326 CRS with coordinates in degrees, and its transform indicated a pixel size of 0.01 degrees. The

new raster uses the EPSG:3857 CRS, commonly known as Web Mercator, with coordinates in meters. The new transform shows a pixel size of approximately 1263.50 meters. This reprojection technique allows us to adjust any raster data file to the desired CRS, aligning the data structure to both the use case and every other data set we are analyzing.

69. Compute Zonal Statistics

Zonal statistics is a geospatial analysis technique used to derive statistical measures of raster data within defined vector zones. By computing zonal statistics, we can summarize raster data, such as population density, elevation, or land cover, over specific areas, such as countries or administrative regions. This technique helps to extract meaningful insights and patterns from complex geospatial datasets.

In this section, we will calculate the total population for each country in Europe using the GHSL population raster dataset and a GeoDataFrame containing the geometries of European countries. We will achieve this by masking the raster data with each country's geometry and then summing the population values within each mask.

The steps involve reading the GeoDataFrame of global countries, clipping it to the area of interest, reprojecting it to match the raster's coordinate reference system (CRS), and performing zonal statistics using Rasterio and The resulting population totals are then added to the country-level GeoDataFrame and visualized using

In

```
1 # Import necessary libraries
2 import geopandas as gpd
3 from shapely.geometry import box
4 import rasterio
5 import numpy as np
6 from rasterio.mask import mask
```

```
7 import matplotlib.pyplot as plt
8
9 # Read the natural earth low resolution dataset
10 gdf =
11
12 # Create a bounding box for Europe
13 bbox =
14 bbox =
15
16 # Clip the GeoDataFrame to Europe and exclude Russia
17 gdf_europe = gdf[gdf['continent'] == 'Europe']
18 gdf_europe = gdf_europe[gdf_europe['geometry'].within(bbox)]
19 gdf_europe = gdf_europe[gdf_europe['country'] != 'Russia']
```

In

```
1 # Function to calculate total population for each country
2 def calculate_population(gdf_countries, raster_file):
3     # Open the raster file
4     with rasterio.open(raster_file) as src:
5         raster_data = src.read()
6         raster_transform = src.transform
7         raster_crs = src.crs
8
9         # Ensure the GeoDataFrame is in the same CRS as the raster
10        gdf_countries = gdf_countries.to_crs(raster_crs)
11
12        total_population = []
13
14        for _, country in gdf_countries.iterrows():
```

```
15      # Mask the raster with the country's geometry

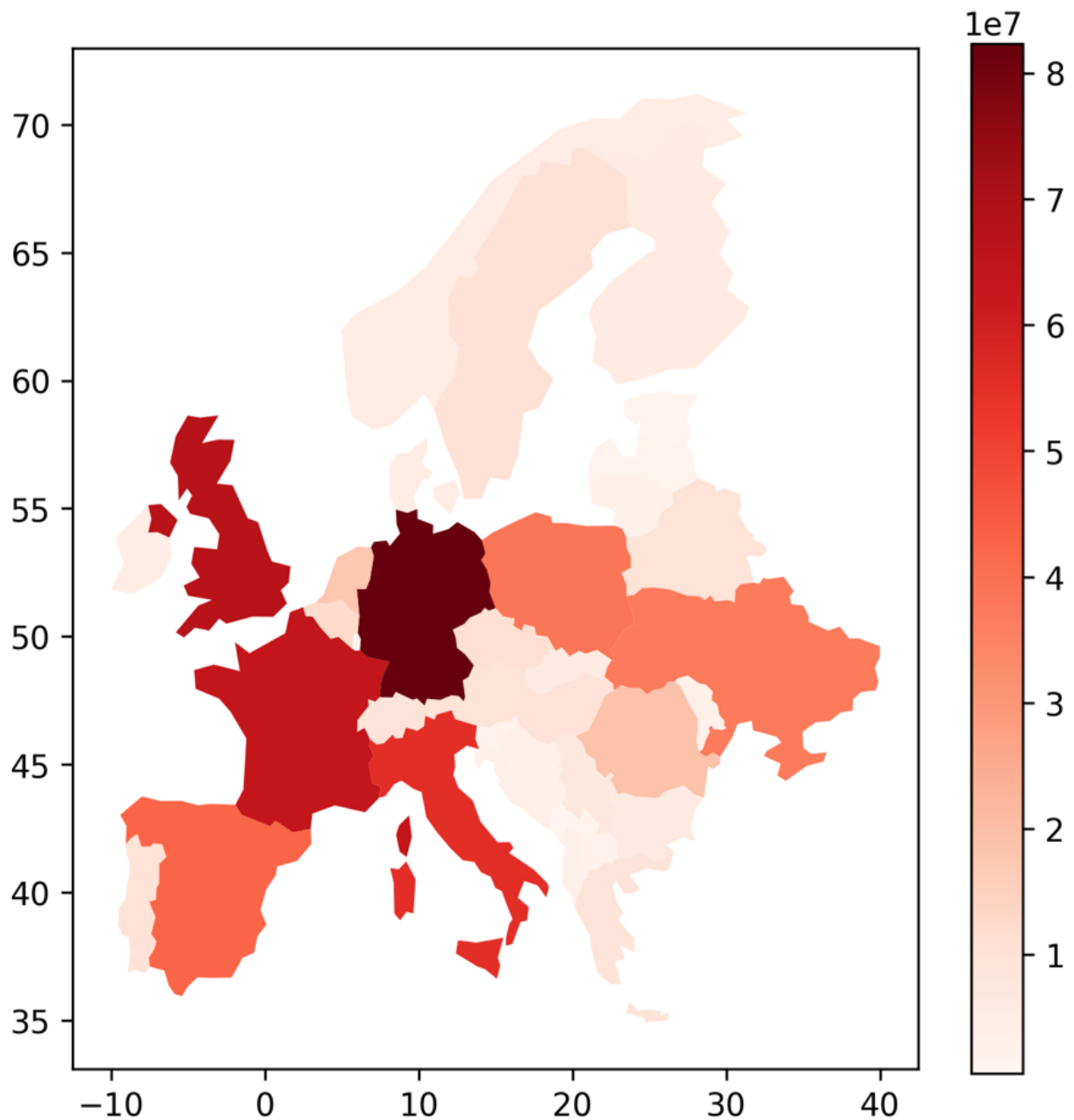
16      mask_array, mask_transform = mask(src,
17
18
19
20      mask_array = mask_array[0]
21
22      # Calculate the total population within the country
23      country_population =
24
25
26      # Add the total population to the GeoDataFrame
27      gdf_countries['total_population'] = total_population
28
29      return gdf_countries
30
31  # Path to your clipped raster file
32  clipped_raster_file = 'clipped_raster_europe.tif'
33
34  # Calculate the total population for each country
35  gdf_europe_with_population = calculate_population(gdf_europe,
36                                                    clipped_raster_file)
37
38  # Display the updated GeoDataFrame
39
```

Out



In

```
1 # Plot the population data on a map
2 f, ax = 1, 6))
3
4
5
6
7
8
```



Zonal statistics allow for the aggregation of raster data within specified vector zones, providing valuable insights into spatial data distributions. This example demonstrates the calculation of the total population per European country by aggregating the GHSL's grid-level population data.

We note that one may expect the population levels to be integer numbers, while here we see floats - the reason behind this is that the GHSL data contains modeled population levels, which were not rounded to integers.

We also note that when we start to compare the computed population levels of each country to official records, in some cases, we might see as high differences as about 10% to today's population, which may have several reasons. First, we are using projected data for 2030, which may include various factors. Second, the GHSL is a global data set, which means that it provides data from every corner of the globe, which usually comes at the cost of local accuracy. Third, our masking and the country-level boundaries may not match the latest official boundary lines perfectly.

70. Convert a Raster Grid into Vector Data

Converting raster data into vector data is a geospatial analytics exercise that allows for more flexibility when switching back and forth between raster and vector data. Raster data is composed of a grid of cells, each with one or more numeric values, while vector data represents features as points, lines, or polygons.

In this section, we will convert the clipped European population raster grid into a vector format - a GeoDataFrame of point geometries created in where each point corresponds to a raster cell with a population value. The process involves reading the raster data using iterating over the raster cells, extracting their coordinates and values, and creating Point geometries for non-NaN and non-zero values. These points and their associated population values are then used to create a GeoDataFrame with a defined coordinate reference system (CRS). Finally, we visualize the resulting point data using

In

```
1 # Import necessary libraries
2 import geopandas as gpd
3 import rasterio
4 from shapely.geometry import Point
5 import matplotlib.pyplot as plt
6 from matplotlib.colors import LogNorm
7
8 # Path to your clipped raster file
```

```

9 clipped_raster_file = 'clipped_raster_europe.tif'
10
11 # Open the raster file and read the data

12 with as src:
13     raster_data =
14     transform =
15     rows, cols =
16
17 # Create lists to hold point geometries and cell values
18 points = []
19 values = []
20
21 # Loop through the raster data
22 for row in range(0, rows, 10): # Adjust the step size as needed
23     for col in range(0, cols, 10): # Adjust the step size as needed
24         value = raster_data[row, col]
25         if value > 0: # Only add non-NaN and non-zero values
26             # Get the (x, y) coordinates for the cell
27             x, y = row, col
28             y))
29

```

In

```

1 # Create a GeoDataFrame
2 gdf = values, 'geometry': points})
3
4 # Use the original CRS of the raster data
5 =

```

6

7 # Print the size of the GeoDataFrame

8 print(len(gdf))

9

10 # Display the GeoDataFrame

11

12

13 # Normalize the population values for better visualization

14 norm =

15

16 # Plot the GeoDataFrame

17 f, ax = 1, 8))

18

19

20

21

22

23

24

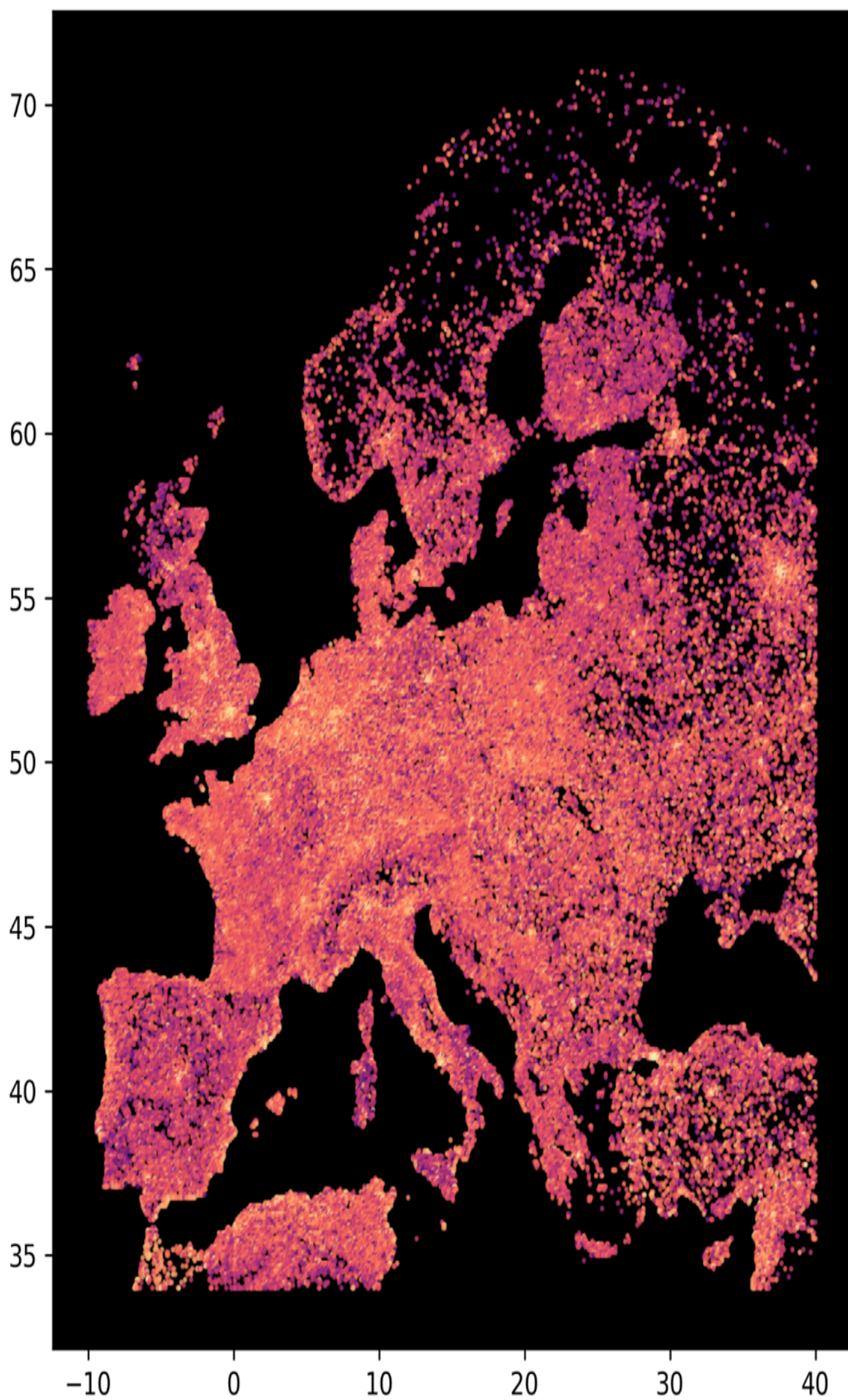
25

26

64907

| | | | |
|-------|-------------------|-------------------|-------------------|
| 64907 | 64907 64907 64907 | 64907 64907 64907 | 64907 64907 64907 |
|-------|-------------------|-------------------|-------------------|

| | |
|-------------------|-------------------|
| 64907 64907 64907 | 64907 64907 64907 |
|-------------------|-------------------|



Converting raster grids into vector data supports more detailed and flexible geospatial analysis. In this example, we reviewed how to transform a population raster grid into a vector format consisting of point geometries located according to the corresponding grid cell's coordinates.

71. Reading Large Raster Files Efficiently

Reading large raster files efficiently is crucial for geospatial analysis, when dealing with high-resolution datasets that can be dozens of gigabytes in size. Efficient data handling prevents excessive memory usage and speeds up processing times. Now we will learn about the library called which provides handy tools to read and manipulate large raster files using chunking and parallel processing.

We will read two large raster files containing global population data for the year 2030, provided by the Global Human Settlement Layer (GHSL) project. These files are at different spatial resolutions, 1000m and 100m pixel size, in Mollweide projection. We use the `rioxarray.open_rasterio` method to parse these files, whereby setting the `chunks` parameter, we can control how the data is divided into smaller pieces for processing while using the `lock` parameter, we can manage access to the data in parallel computing environments. These optimizations enable handling large datasets without overwhelming system resources.

In

```
1 # Import rioxarray
2 import rioxarray
3
4 # Define file paths for the raster files
5 folder_100 = 'GHS_POP_E2030_GLOBE_R2023A_54009_100_V1_0'
6 file_name_100 =
'GHS_POP_E2030_GLOBE_R2023A_54009_100_V1_0.tif'
7 raster_file_100 = folder_100 + '/' + file_name_100
```



```
8
9 folder_1000 = 'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0'
10 file_name_1000 =
'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0.tif'
11 raster_file_1000 = folder_1000 + '/' + file_name_1000
```

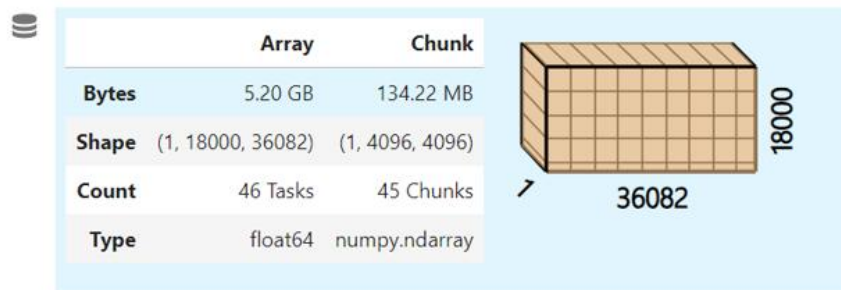
In

```
1 # Read the 1000m resolution raster file using rioxarray
2 data_array_1000 =
3
4
5 data_array_1000
```

Out

xarray.DataArray

```
1
y: 18000
x: 36082
```



▼ Coordinates:

| | | | | | |
|-------------|--------|---------|-------------------------------------|--|--|
| band | (band) | int64 | 1 | | |
| x | (x) | float64 | -1.804e+07 -1.804e+07 ... 1.804e+07 | | |
| y | (y) | float64 | 9e+06 8.998e+06 ... -9e+06 | | |
| spatial_ref | () | int64 | 0 | | |

▼ Attributes:

AREA_OR_POINT : Area
 STATISTICS_MAX... 452635.11543083
 STATISTICS_MEA... 61.892180872876
 STATISTICS_MINI... 0
 STATISTICS_STD... 767.17038013196
 STATISTICS_VALI... 21.26
 _FillValue : -200.0
 scale_factor : 1.0
 add_offset : 0.0

In

```

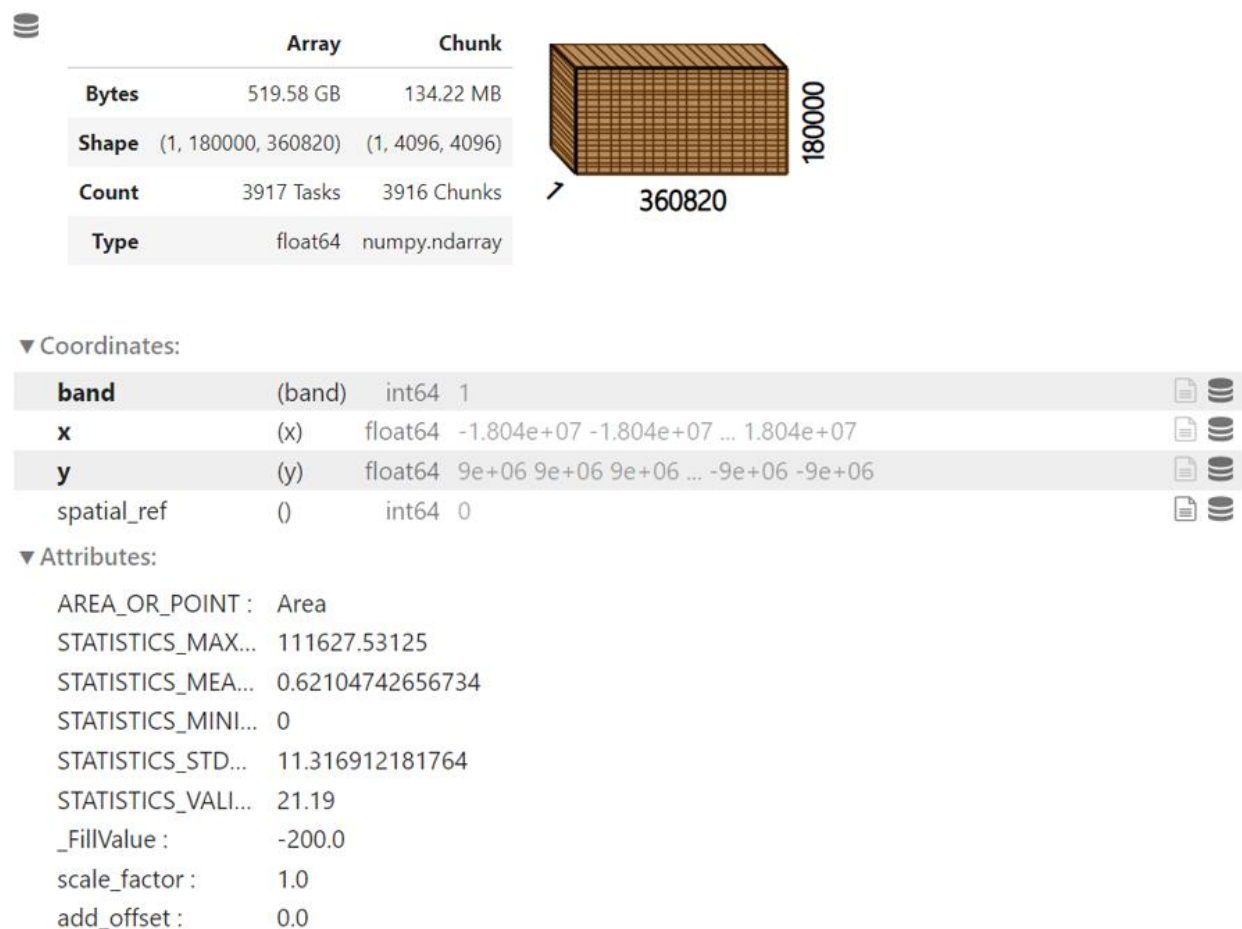
1 # Read the 100m resolution raster file using rioxarray
2 data_array_100 =
3
4
5 data_array_100

```

Out

xarray.DataArray

1
y: 180000
x: 360820



After reading and printing the attributes of these two data files of 1000m and 100m resolution, we can observe that both files have a single band, and the 1000m resolution raster data has dimensions of (1, 18000, 36082) with a maximum population value of 452,635 and a mean of approximately 61.89, indicating coarser granularity suitable for large-scale analysis. In contrast, the 100m resolution raster data, with dimensions ten times larger than (1, 180000, 360820), shows a maximum population value of 111,627 and a significantly lower mean of around 0.62, reflecting finer detail and more precise population distribution, suitable for detailed, small-scale analysis.

72. Clipping Large Raster Files

Clipping raster files is a common geospatial operation used to focus on a specific region of interest, thereby reducing data size and improving processing efficiency. While we previously demonstrated clipping using `we` we now introduce a new tool, which offers advanced and efficient data handling capabilities for large datasets. Clipping with `xarray` leverages its integration with providing seamless workflows for large-scale geospatial data analysis. The `xarray` library, along with facilitates efficient raster data operations with the benefits of chunking and parallel processing.

In this section, we will clip a large raster file to a defined bounding box using `xarray` and The process involves reading the raster data into an `xarray.DataArray` and then using the `rio.clip_box` method to specify the bounding box coordinates. We then compare the size of the original and clipped data arrays to observe the effect of clipping. Again, we used Europe as a target area, defining its bounding box in the Mollweide coordinate system.

In

```
1 # Import necessary libraries
2 import xarray as xr
3 import rioxarray
4
5 # Define file path for the raster file
6 folder_1000 = 'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0'
7 file_name_1000 =
'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0.tif'
```

```
8 raster_file_1000 = folder_1000 + '/' + file_name_1000
9
10 # Read the raster file using rioarray with chunking and lock
11 data_array =
12
13
14
15 # Clip the data array to the defined bounding box
16 data_array_c =
17
18
19
20
21 # Convert clipped data array to xarray DataArray
22 data_array_c =
23
24 # Print the size of the original and clipped data arrays
25 print(f'Original DataArray size:
26 print(f'Clipped DataArray size:
27
28 # Display the original data array
29 data_array
```

Original DataArray size: 649476000
Clipped DataArray size: 12250000

Out

xarray.DataArray

1

y: 18000

x: 36082

| | Array | Chunk |
|-------|-------------------|-----------------|
| Bytes | 5.20 GB | 134.22 MB |
| Shape | (1, 18000, 36082) | (1, 4096, 4096) |
| Count | 46 Tasks | 45 Chunks |
| Type | float64 | numpy.ndarray |

18000
36082

▼ Coordinates:

| | | | | | |
|-------------|--------|---------|-------------------------------------|--|--|
| band | (band) | int64 | 1 | | |
| x | (x) | float64 | -1.804e+07 -1.804e+07 ... 1.804e+07 | | |
| y | (y) | float64 | 9e+06 8.998e+06 ... -9e+06 | | |
| spatial_ref | () | int64 | 0 | | |

▼ Attributes:

AREA_OR_POINT :

Area

STATISTICS_MAX...

452635.11543083

STATISTICS_MEA...

61.892180872876

STATISTICS_MINI...

0

STATISTICS_STD...

767.17038013196

STATISTICS_VALI...

21.26

_FillValue :

-200.0

scale_factor :

1.0

add_offset :

0.0

In

```
1 # Display the clipped data array
2 data_array_c
```

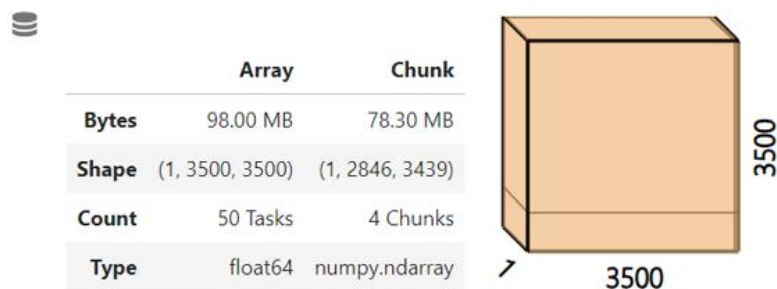
Out

xarray.DataArray

1

y: 3500

x: 3500



▼ Coordinates:

| | | | | | |
|-------------|--------|---------|-----------------------------------|--|--|
| band | (band) | int64 | 1 | | |
| x | (x) | float64 | -9.995e+05 -9.985e+05 ... 2.5e+06 | | |
| y | (y) | float64 | 7.75e+06 7.748e+06 ... 4.250e+06 | | |
| spatial_ref | () | int64 | 0 | | |

▼ Attributes:

AREA_OR_POINT : Area
STATISTICS_MAX... 452635.11543083
STATISTICS_MEA... 61.892180872876
STATISTICS_MINI... 0
STATISTICS_STD... 767.17038013196
STATISTICS_VALI... 21.26
scale_factor : 1.0
add_offset : 0.0
_FillValue : -200.0

The original DataArray has dimensions of (1, 18000, 36082) while the clipped one is the size of (1, 3500, 3500), which illustrates the results of clipping the global data file to contain only Europe.

73. Visualizing Large Raster Files

Visualizing large raster files is essential for understanding and analyzing the geospatial data stored in that file, especially when dealing with high-resolution datasets that contain vast amounts of information. Traditional plotting methods can be inefficient for such large datasets, so tools like [datashader](#) and [colorcet](#) are used for efficient and visually appealing data visualizations. `datashader` efficiently renders large datasets, while `colorcet` provides a wide range of [colormaps](#) to enhance the visual representation of the data.

In this section, we will first read the global GHSL data at 1000m resolution and clip it to cover Europe only. Then, we conduct additional data processing steps, such as ensuring all values are positive and are oriented in the correct direction. We then create a canvas with define the plot size and aspect ratio, rasterize the data, and apply a colormap from `colorcet` for visual enhancement. Finally, we use `datashader.transfer_functions` to display the raster image.

In

```
1 # Import necessary libraries
2 import datashader as ds
3 from colorcet import palette
4 from datashader import transfer_functions as tf
5 import xarray as xr
6 import numpy as np
7 import rioxarray
8
9
```



```

10 # Define file path for the raster file

11 folder_1000 = 'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0'
12 file_name_1000 =
'GHS_POP_E2030_GLOBE_R2023A_54009_1000_V1_0.tif'
13 raster_file_1000 = folder_1000 + '/' + file_name_1000
14
15 # Read and clip the raster file using rioxtarray with
16 data_array =
17
18
19 data_array_c =
20
21
22
23 data_array_c =
24
25 # Transform the raster
26 data_array_c =
27 data_array_c = > 0)
28 data_array_c =
29 data_array_c = 0)
30
31 # Get the image size
32 size = 800
33 aspect_ratio = /
34
35 # Create the datashader canvas

36 cvs = * size))
37 raster =
38
39 # Draw the image

```

```
40 cmap = palette["kgy"]
41 img =
42 img = "black")
43 img
```

Out



74. Downsampling a Large Raster File

Downsampling is a technique used to reduce the resolution of raster data, making it more manageable for analysis and visualization. This technique is particularly useful when developing initial explorative analytics on a new, large-scale data set in preparation for the final computation conducted on the full database. The main idea behind this process involves aggregating the values of neighboring cells to create a lower-resolution raster, which retains essential information while significantly reducing data size.

In this section, we will demonstrate how to downsample the 100m resolution GHSL population data file using After importing the necessary libraries, we use the usual raster data loader. Then, we define a `downsample_raster` function that takes a `data_array` and a factor as inputs and returns the downsampled raster by averaging the values of pixels within the specified factor. For this, we define different downsampling levels and apply the downsampling function to the original data array with these factor values to finish the downsampling. At each downsampling step, we also print the number of remaining grid cells to illustrate the efficiency of downsampling.

In

```
1 # Library imports
2 import xarray as xr
3 import rioxarray
4 import numpy as np
```

5

6 # Define file path for the raster file

7 folder_100 = 'GHS_POP_E2030_GLOBE_R2023A_54009_100_V1_0'

8 file_name_100 =

'GHS_POP_E2030_GLOBE_R2023A_54009_100_V1_0.tif'

9 raster_file_100 = folder_100 + '/' + file_name_100

10

11 # Read the raster file using rioxtarray with chunking and lock

12 data_array =

13

14

15

16 # Downsampling function

17 def downsample_raster(data_array, factor):

18 return

19

20 # Different levels of downsampling

21 downsampling_factors = [2, 5, 10, 20, 100]

22

23 # Perform downsampling and print the number of cells (KPI)

24 for factor in downsampling_factors:

25 downsampled_data_array = downsample_raster(data_array, factor)

26 num_cells =

27 print(f'Downsampling Factor: {factor}, Number of Cells:

{num_cells}")

Downsampling Factor: 2, Number of Cells: 16236900000

Downsampling Factor: 5, Number of Cells: 2597904000

Downsampling Factor: 10, Number of Cells: 649476000

Downsampling Factor: 20, Number of Cells: 162369000

Downsampling Factor: 100, Number of Cells: 6494400

This example shows how to downsample a raster dataset at various levels, with significant reductions in the number of cells as the downsampling factor increases: starting from the original size, a downsampling factor of 2 reduces the cell count to 16,236,900,000, while a factor of 100 reduces it further to 6,494,400 cells.

Summary on Raster Data

In this chapter, we explored various techniques for managing and analyzing raster data using Python. We began by reading and writing raster files with Rasterio, then demonstrated clipping rasters to specific regions using GeoPandas. Reprojecting rasters was covered to facilitate coordinate system transformations, and we explored visualization methods using Rasterio and Matplotlib.

We enhanced raster visualization with histogram equalization and applied simple functions like normalization and reclassification. We computed zonal statistics to summarize raster data within vector zones and converted raster grids into vector data, allowing for more flexible analysis. We learned about efficient reading and manipulation of large raster files with Rioxarray and Xarray. Lastly, we highlighted the importance of downsampling large raster files to reduce data size while retaining essential information.

These techniques equip us with the skills to handle extensive raster datasets effectively, enabling sophisticated geospatial analyses and streamlined data processing workflows.

Introduction to OpenStreetMap Data



OpenStreetMap (OSM) is a comprehensive and freely available crowd-sourced geospatial data platform, offering detailed information on various

geographic features such as administrative boundaries, points of interest (POIs), parks, building footprints, and road networks. Leveraging this type of data is essential for urban planning, resource allocation, environmental studies, and transportation analysis, which made OSM one of the most standard data sources of our profession.

In this chapter, we will explore how to use Python to download and visualize different types of OpenStreetMap data. We will start by querying administrative areas, such as countries, cities, and districts, and then visualize them on a map. Next, we will learn about downloading and visualizing POIs, including amenities like cafes and gift shops. We will then move on to extracting and displaying park polygons and building footprints, providing insights into urban green spaces and infrastructure. Finally, we will demonstrate how to download and examine road networks, which are crucial for transportation planning and urban connectivity studies.

75. Downloading Administrative Areas from OpenStreetMap (OSM)

Administrative areas such as countries, cities, and districts are essential for various geospatial analyses, including urban planning, resource allocation, and demographic studies. [OpenStreetMap \(OSM\)](#) provides a rich source of such geospatial data, while the Python library [OSMnx](#) provides easily-to-use methods to retrieve them.

In this section, we will demonstrate how to obtain administrative areas from OSM using As an example, we will extract and visualize the country boundaries for Hungary, the city boundaries of Budapest, and the district boundaries of 1st district of Budapest, and then combine these boundaries into a single plot.

This process involves downloading the administrative boundaries for each area using the `geocode_to_gdf` function from which converts place names into GeoDataFrames. After plotting the individual boundaries for visual inspection, we will merge the GeoDataFrames into a single one for a combined visualization. Finally, we will enhance the plot with a customized legend to clearly differentiate between the administrative levels.

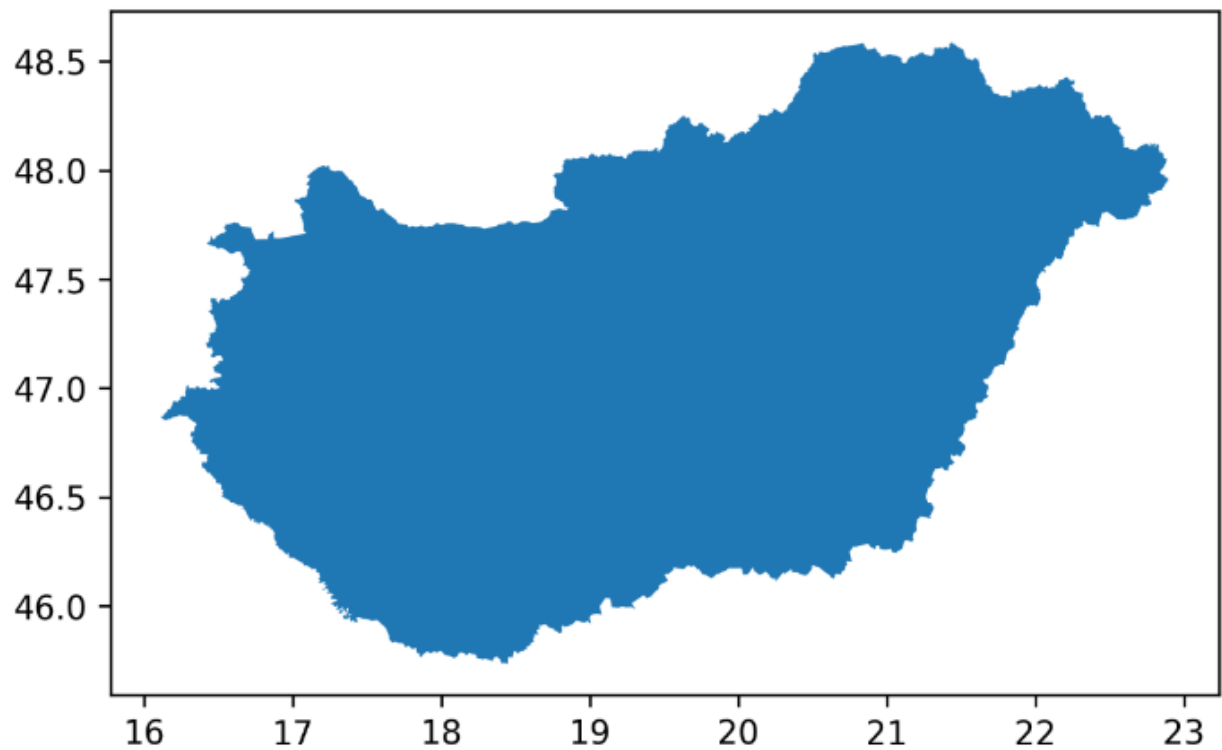
In

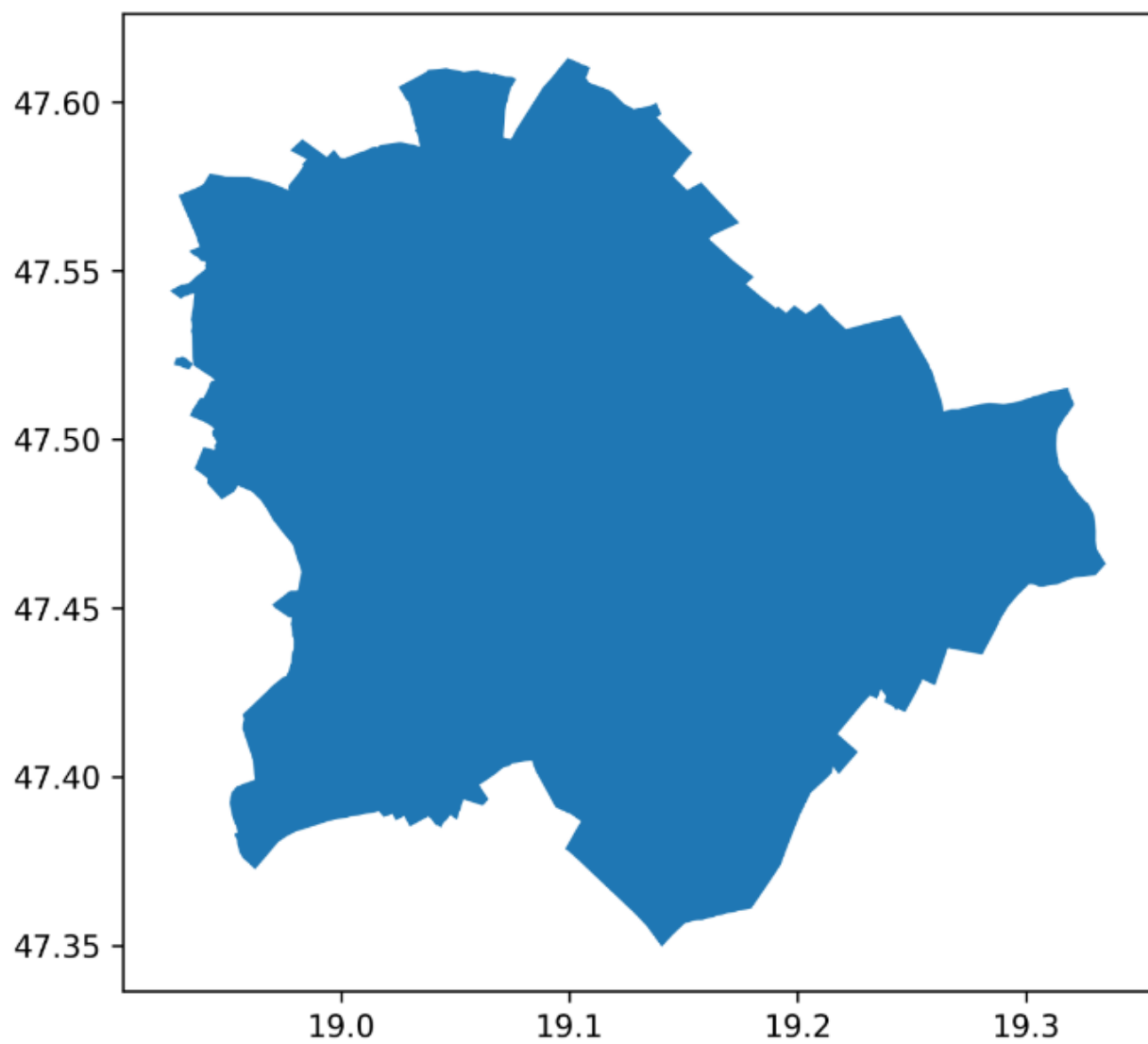
```
1 #Import all the libraries we use
2 import osmnx as ox
3 import pandas as pd
4 import matplotlib.pyplot as plt
5
6 # Download the administrative boundary of Hungary
```

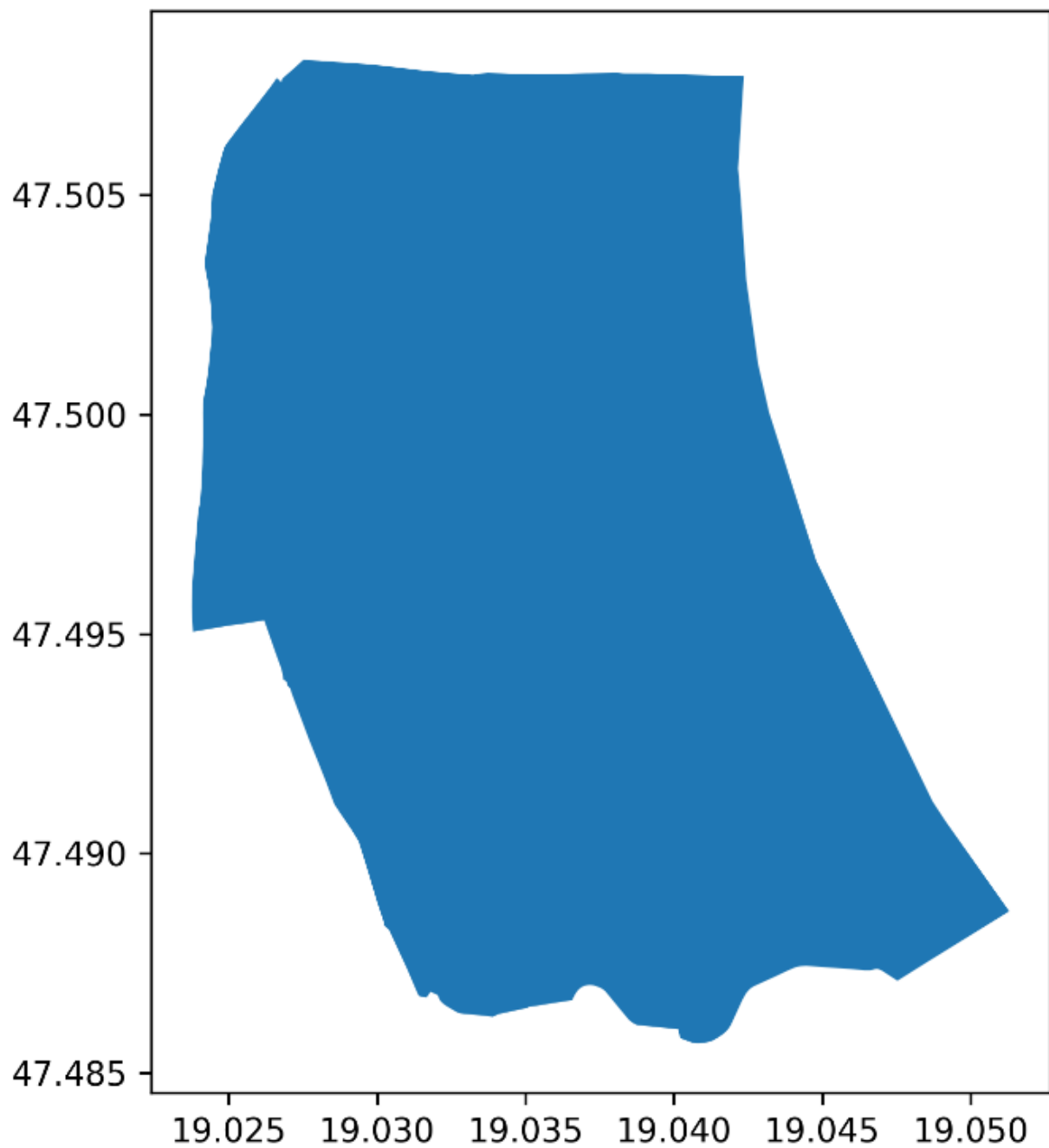
```
7 admin_country =  
  
8  
9  
10 # Download the administrative boundary of Budapest  
11 admin_city =  
12  
13  
14 # Download the administrative boundary of the 1st district of Budapest  
15 admin_district = district, Budapest')  
16
```

Out

>







In

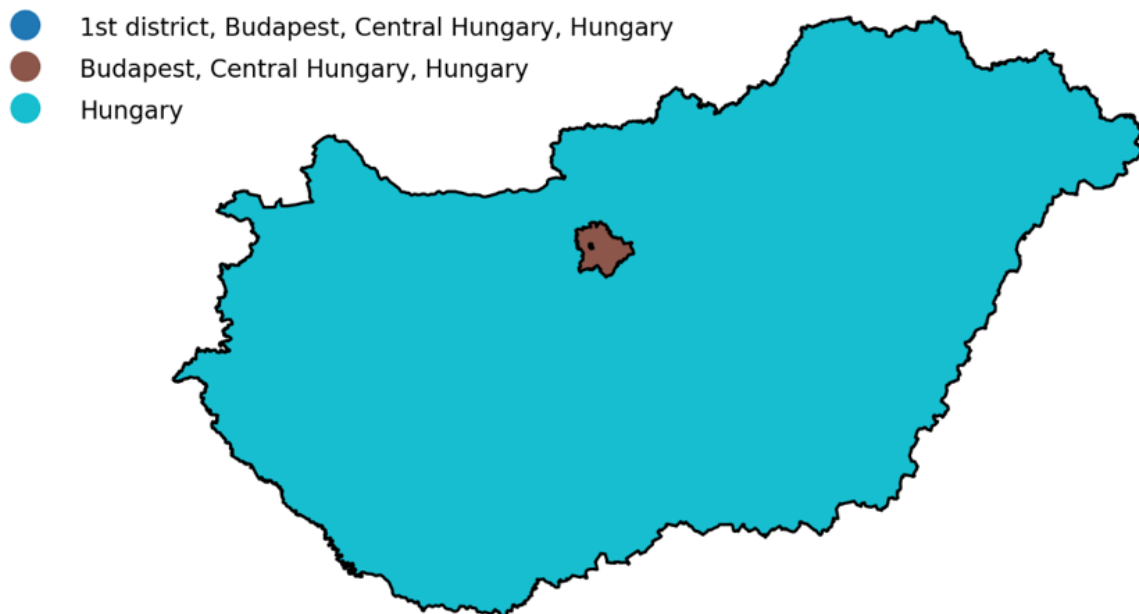
```
1 # Combine all the administrative boundaries into a single GeoDataFrame
2 admin_all = admin_city, admin_district])
3
4 # Create a plot to visualize the combined administrative boundaries
```

```

5 f, ax = 1, 4))

6
7
8 # Customize the legend
9 legend =
10 1)) # Position the legend outside the plot
11 # Turn off the frame
12 for text in
13     # Change the font size
14 1.0)) # Place legend outside the plot
15
16 # Turn off the axis
17 # Display the plot

```



Retrieving and visualizing administrative areas from OpenStreetMap is a straightforward process using OSMnx that provides a valuable base layer for

various geospatial analytics tasks, such as capturing spatial relations and characterizing specific areas.

76. Using OSMnx to Download Points of Interest (POIs)

Points of Interest (POIs) are crucial for various geospatial analyses, urban planning tasks, and location-based services. POIs refer to entities on a map described by a pair of coordinates and may include amenities such as cafes, restaurants, and shops, which provide valuable insights into the characteristics and functionalities of an area. OpenStreetMap (OSM) is an excellent source of free POI data, and the osmnx library makes it easy to retrieve and visualize these points.

In this section, we will demonstrate how to use osmnx to download POIs within a specified administrative boundary. As an example, we will extract and visualize POIs for cafes and gift shops in the 1st district of Budapest. This process involves using the `features_from_polygon` function to query POIs within the given boundary polygon and then plotting the results. From the [OSM Wiki](#) site we can select the relevant tags for our use-case, such as matching cafe from the amenity category and gift from the shop category.

In

```
1 # Import osmnx and matplotlib
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4
5 # Define the polygon for the administrative
6 # boundary of the 1st district of Budapest
7 admin_poly =
8
```

```

9 # Download POIs for cafes within the administrative boundary
10 cafes = 'cafe'})
11 print("Number of cafes:", len(cafes))
12 print("Type of data:", type(cafes))
13

```

Number of cafes: 47

Type of data: 'geopandas.geodataframe.GeoDataFrame'>

Out

| element_type | osmid | address | address:code | address:street | amenity | contact:email | contact:facebook | contact:website | indoor_seating | name | outdoor_seating | ... | payment:coins | payment:contactless | payment:moneta | payment:... |
|--------------|-----------|---------|--------------|----------------|---------|----------------------------|-------------------|------------------------------------|----------------|-----------|-----------------|-----|---------------|---------------------|----------------|-------------|
| node | 344907128 | Badsgen | 111 | Comma 11 | cafe | ca.vincelofficel@gmail.com | LevinCafeFacebook | https://www.facebook.com/LevinCafe | yes | LevinCafe | yes | ... | NaN | NaN | NaN | NaN |
| | 735220069 | Badsgen | 114 | Disa 11 | cafe | NaN | NaN | NaN | NaN | Go.com | NaN | ... | NaN | NaN | NaN | NaN |
| | 735221987 | Badsgen | 114 | Disa 11 | cafe | NaN | NaN | NaN | NaN | LevinCafe | NaN | ... | NaN | NaN | NaN | NaN |
| | 735222199 | Badsgen | 114 | Disa 11 | cafe | NaN | NaN | NaN | yes | LevinCafe | yes | ... | NaN | NaN | NaN | NaN |
| | 735225266 | NaN | NaN | NaN | cafe | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN |

5 rows x 17 columns

In

```

1 # Download POIs for gift shops within the administrative boundary
2 gift_shops = 'gift'})
3 print("Number of gift shops:", len(gift_shops))
4 print("Type of data:", type(cafes))
5

```

Number of gift shops: 14

Type of data: 'geopandas.geodataframe.GeoDataFrame'>

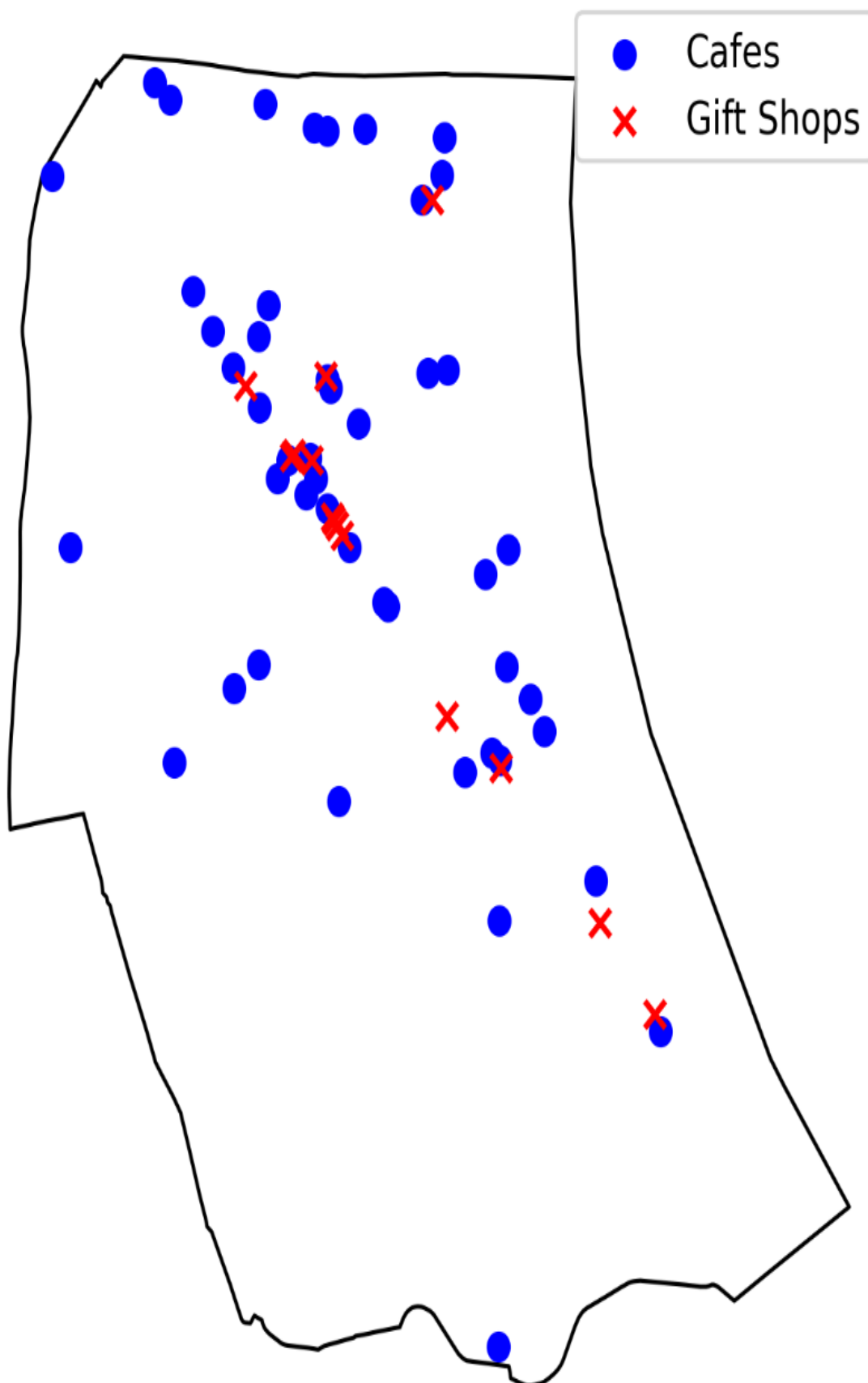
Out

| element_type | osmid | addr:city | addr:house:number | addr:post:code | addr:street | email | name | operator | phone | shop | geometry | ... | dog | free_refill | website | wheelchair | level | denomination | desc |
|--------------|------------|-----------|-------------------|----------------|--------------------|----------------------|-----------------|----------------------------------|----------------|------|---------------------------|-----|-----|-------------|--------------------------------|------------|-------|--------------|------|
| node | 3392225658 | Budapest | 5 | 1014 | Szentharomság utca | hadikshop@herend.com | Hadik márkabolt | Herendi Porcelánmanufaktúra Zrt. | +36 1 225 1051 | gift | POINT (19.03310 47.50137) | — | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 3508763494 | Budapest | NaN | 1011 | Markovits hán utca | NaN | Bikesziget | NaN | NaN | gift | POINT (19.03763 47.50565) | — | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 5039257227 | NaN | NaN | NaN | NaN | NaN | Prezent | NaN | NaN | gift | POINT (19.04492 47.49193) | — | yes | yes | http://www.prezentbudapest.hu/ | no | NaN | NaN | NaN |
| | 5842060876 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | gift | POINT (19.03810 47.49696) | — | NaN | NaN | NaN | NaN | NaN | 0 | NaN |
| | 5842060877 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | gift | POINT (19.03989 47.49608) | — | NaN | NaN | NaN | no | 0 | NaN | NaN |

5 rows × 21 columns

In

```
1 # Create a plot to visualize the administrative boundary and POIs
2 f, ax = 1, 6))
3
4 # Plot the administrative boundary
5
6
7 # Plot the cafes and gift shops
8
9 Shops')
10
11 # Customize the plot
12
13
14
```



In this example, we briefly illustrated how to query POI data from OSM within a given target area. Namely, we were able to obtain 47 cafes and 14 gift stores with their locational information stored in GeoDataFrames, and supplement the downloading process by a quick visual as well.

77. Using OSMnx to Download Parks as Polygons

Parks and green spaces are vital components of urban environments, contributing to the quality of life and ecological balance. Identifying and analyzing these areas can provide valuable quantitative insights for urban planning, environmental studies, and public health assessments.

OpenStreetMap (OSM) offers comprehensive data on parks, and the `osmnx` library allows us to easily retrieve and visualize these green spaces.

In this section, we will extract the park polygons in the 1st district of Budapest and visualize these areas on a map. This process involves using the `features_from_polygon` function to query park polygons within the given boundary polygon and then plotting the results.

In

```
1 # Importing all libraries we use
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4
5 # Define the polygon of the 1st district of Budapest
6 admin_poly =
7
8 # Download park polygons within the administrative boundary
9 parks = 'park'})
10 print("Number of parks:", len(parks))
11
```

Number of parks: 51

Out

| | | geometry | access | name | wheelchair | operator | nodes | layer | leisure | name:de | wikidata | ... | alt_name | name:etymology:wikidata | source:name | source | esperanto | name:en | name:eo | name:hu | ways | type |
|--------------|----------|--|---------|-------------------|------------|-------------------|--|-------|---------|-------------------------------|----------|-----|----------|-------------------------|-------------|--------|-----------|---------|---------|---------|------|------|
| element_type | osmid | | | | | | | | | | | | | | | | | | | | | |
| way | 15004290 | POLYGON [[19.02444, 47.50432, 19.02441, 47.50430,... | NaN | Vérmező | NaN | NaN | [148145988, 10686553844, 6593030182, 659303018... | -2 | park | Blutfeld | Q1462614 | ... | NaN | | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 32612302 | POLYGON [[19.03866, 47.49067, 19.03877, 47.49070,... | NaN | NaN | NaN | NaN | [11933116830, 11933116809, 11933116810, 119331... | NaN | park | NaN | NaN | ... | NaN | | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 37135055 | POLYGON [[19.03056, 47.50468, 19.03050, 47.50450,... | NaN | Bécsi kapu tér | NaN | NaN | [3547379781, 10997076311, 3547379776, 10997076... | NaN | park | Platz des Wiener Toners | Q299969 | ... | NaN | | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 37135100 | POLYGON [[19.03771, 47.49925, 19.03780, 47.49927,... | private | NaN | NaN | Miniszterelnökség | [832490937, 370517103, 8728469582, 8728469584... | NaN | park | NaN | NaN | ... | NaN | | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| | 85552126 | POLYGON [[19.03724, 47.50057, 19.03720, 47.50053,... | NaN | NaN | NaN | NaN | [8647966714, 8647966713, 8647966715, 864796671... | NaN | park | NaN | NaN | ... | NaN | | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

5 rows × 27 columns

In

- 1 # Create a plot to visualize the admin boundary and park polygons
- 2 f, ax = 1, 6))
- 3
- 4 # Plot the administrative boundary
- 5
- 6
- 7 # Plot the parks
- 8
- 9
- 10 # Customize the plot
- 11
- 12



download and visualize parks as polygons from OpenStreetMap is a powerful way to enhance geospatial analyses related to urban green spaces and, gain valuable insights into the distribution and extent of green areas and use these as inputs for advanced spatial analysis.

78. Using OSMnx to Download Building Footprints

Building footprints provide vital information for urban planning, infrastructure development, and environmental studies. By analyzing building footprints, we can gain quantitative insights into urban density, architectural styles, and land use patterns. OpenStreetMap (OSM) contains detailed data on building footprints, and the osmnx library makes it straightforward to download and visualize this information.

In this section, we will demonstrate how to use osmnx to download building footprints within the 1st district of Budapest and visualize these footprints on a map. This process involves using the `features_from_polygon` function to query building footprints within the given boundary polygon, and Matplotlib to plot the results.

In

```
1 # Import osmnx for and matplotlib
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4
5 # Define the polygon for the administrative boundary of the 1st district
  of Budapest
6 admin_poly =
7
8 # Download building footprints within the administrative boundary
9 footprints =
10 print("Type of the footprints data:", type(footprints))
```



```
11 print("Number of buildings:", len(footprints))

12
13 # Create a plot to visualize the administrative boundary and building
footprints
14 f, ax = plt.subplots(1, 8))
15
16 # Plot the administrative boundary
17
18
19 # Plot the building footprints
20
21         cmap = 'Greys',
22
23
24         linewidth = 0.5)
25
26 # Customize the plot
27
28
```

Type of the footprints data: 'geopandas.geodataframe.GeoDataFrame'>
Number of buildings: 1611



Using OSMnx to download and visualize building footprints from OpenStreetMap is an effective way to enhance geospatial analyses related to urban development and land use. By extracting building footprints within a specified area, we can gain valuable insights into urban density and architectural patterns.

79. Using OSMnx to Download Road Networks

Road networks are fundamental components of urban infrastructure, playing a crucial role in transportation planning and navigation systems. Analyzing road networks can provide insights into traffic patterns, connectivity, and accessibility. OpenStreetMap (OSM) offers detailed data on road networks, while the osmnx library was specifically designed to download and visualize these networks effectively.

In this section, we will demonstrate how to use osmnx to download road networks within the 1st district of Budapest and examine the type and size of these graph files. This process involves using the `graph_from_polygon` function to query the road network within a given boundary polygon and then exploring the resulting graph. We also set and tested which allowed us to query data specific to different means of transportation, namely, driving, biking, and walking. Later, we will further explore the details of these road networks in the chapter covering spatial networks.

In

```
1 # Import osmnx for geospatial data from OpenStreetMap
2 import osmnx as ox
3
4 # Define the polygon for the 1st district of Budapest
5 admin_poly =
6
7 # Download the road network for all transport modes
8 G_all =
```

```

9 print("Type of the road network graph (all modes):", type(G_all))
10 print("Number of nodes (all modes):",
11 print("Number of edges (all modes):",
12
13 # Download the road network for walking within the administrative
boundary
14 G_walk =
15 print("\nType of the road network graph (walk):", type(G_walk))
16 print("Number of nodes (walk):",
17 print("Number of edges (walk):",
18
19 # Download the road network for driving within the administrative
boundary
20 G_drive =
21 print("\nType of the road network graph (drive):", type(G_drive))
22 print("Number of nodes (drive):",
23 print("Number of edges (drive):",
24
25 # Download the road network for biking within the administrative
boundary
26 G_bike =
27 print("\nType of the road network graph (bike):", type(G_bike))
28 print("Number of nodes (bike):",
29 print("Number of edges (bike):",

```

```

Type of the road network graph (all modes):
'networkx.classes.multidigraph.MultiDiGraph'>
Number of nodes (all modes): 2024

```

Number of edges (all modes): 5436

Type of the road network graph (walk):

`'networkx.classes.multidigraph.MultiDiGraph'>`

Number of nodes (walk): 1909

Number of edges (walk): 5684

Type of the road network graph (drive):

`'networkx.classes.multidigraph.MultiDiGraph'>`

Number of nodes (drive): 359

Number of edges (drive): 706

Type of the road network graph (bike):

`'networkx.classes.multidigraph.MultiDiGraph'>`

Number of nodes (bike): 622

Number of edges (bike): 1254

Following the logic of the previous sections, here we learned to use OSMnx to download road networks from OpenStreetMap to enhance geospatial analyses related to urban infrastructure and transportation planning. In this example, we used one of the central districts of Budapest and queried its complete road network as well as road network variants designated for walking, driving, and biking. For each variation, we printed the total number of nodes (intersections) and edges (road segments) within each road network.

80. Visualizing Complex Urban Areas

Urban planning and analysis require detailed visualizations to understand the spatial distribution of various elements within a city. Visualizing complex urban scenarios involves displaying multiple layers of geographic data, such as administrative boundaries, building footprints, parks, and points of interest (POIs), on a base map. While in the previous sections, we have individually downloaded data layers like these using `contextily`, now we are now creating a comprehensive visualization combining these different dimensions.

In particular, we will visualize the 1st district of Budapest by combining multiple geographic layers, starting with the administrative boundaries of the districts, enriched by the location of local parks, all the available building footprints, and a variety of POIs essential to daily life.

Additionally, we will overlay these layers on a base map using the previously introduced `contextily` for clear and informative urban visualization. Additionally, we convert all `GeoDataFrames` to the local CRS of Hungary, which we also saw in the Map Projections chapter, EPSG23700, to better align with the local context.

In

```
1 # Import necessary libraries
2 import osmnx as ox
3 import geopandas as gpd
4 import matplotlib.pyplot as plt
5 import contextily as ctx
```


6

7 # Download the administrative boundary of the 1st district of Budapest

8 admin_district = district, Budapest')

9 admin_poly =

10

11 # Download building footprints within the administrative boundary

12 footprints =

13

14 # Download parks within the administrative boundary

15 parks = 'park'}})

16

17 # Download cafes within the administrative boundary

18 cafes = 'cafe'}})

19

20 # Download additional POI types within the administrative boundary

21 restaurants =

22 'restaurant'}})

23 schools =

24 'school'}})

25 hospitals =

26 'hospital'}})

27 bus_stops =

28 'bus_stop'}})

29 parking_areas =

30 'parking'}})

31

32

33 print('Number of restaurants:', len(restaurants))

34 print('Number of schools:', len(schools))

35 print('Number of hospitals:', len(hospitals))

```
36 print('Number of bus_stops:', len(bus_stops))
37 print('Number of parking_areas:', len(parking_areas))
```

Number of restaurants: 72
Number of schools: 12
Number of hospitals: 1
Number of bus_stops: 74
Number of parking_areas: 51

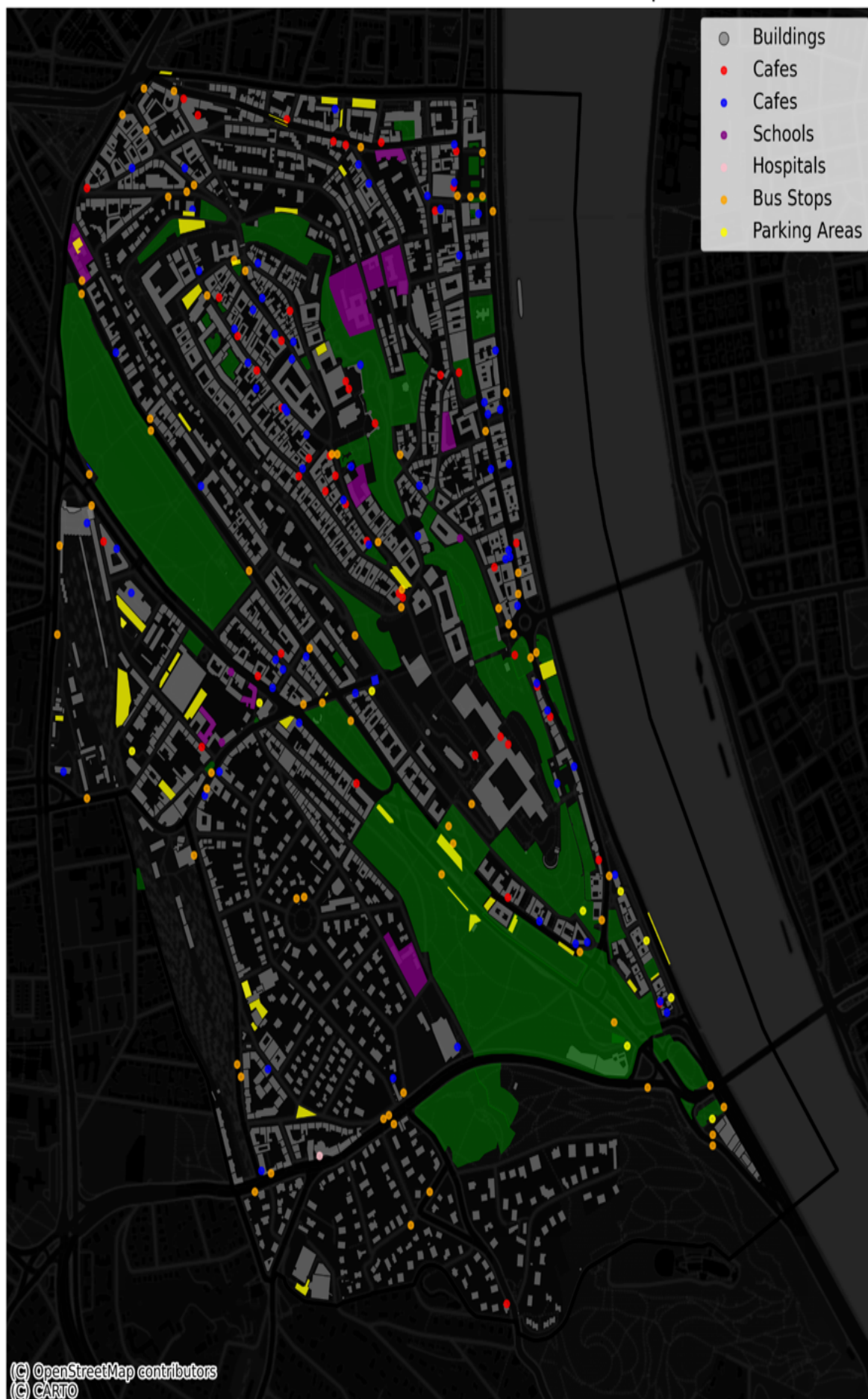
In

```
1 # Create a plot to visualize the administrative boundary and various
  features
2 f, ax = 1, 12))
3
4 # Define a poi plotter function
5 def plot_pois(pois, color, ax, crs, label, zorder):
6
7
8
9
10
11
12
13 # Define the local CRS
14 crs = 23700
15
16 # Plot the administrative boundary
```

```
17
18
19
20
21         Boundary')
22
23 # Plot the building footprints
24
25
26
27
28
29
30
31 # Plot the parks
32
33
34 # Plot the pois
35 plot_pois(cafes, 'red', ax, crs, 'Cafes',
36 plot_pois(restaurants, 'blue', ax, crs, 'Cafes',
37 plot_pois(schools, 'purple', ax, crs, 'Schools',
38 plot_pois(hospitals, 'pink', ax, crs, 'Hospitals',
39 plot_pois(bus_stops, 'orange', ax, crs, 'Bus Stops',
40 plot_pois(parking_areas, 'yellow', ax, crs, 'Parking Areas',
41
42 # Add the basemap from contextily
43 crs = crs, url =
44
45 # Add legend
46 right')
47
```

```
48 # Add title and axis labels
49 of the 1st District of Budapest',
50
51
52
53
54 # Show plot
55
```

Visualization of the 1st District of Budapest



In this section, we combined various geographic layers to visualize the 1st district of Budapest. By overlaying administrative boundaries, building footprints, parks, cafes, restaurants, schools, hospitals, bus stops, and parking areas on a CartoDB Dark Matter basemap, we created a detailed and informative visualization of this urban area. We note that while we downloaded parking areas in the batch of POIs, the resulting data shows they are often stored on OSM as polygons. This comprehensive approach allows for a better understanding and analysis of urban scenarios, aiding in effective urban planning and management.

Summary on OpenStreetMap

In this chapter, we utilized the OSMnx library to access and visualize various types of geospatial data from OpenStreetMap. We began by retrieving administrative boundaries for Hungary, Budapest, and the 1st district of Budapest, combining these into a single visualization. We then extracted and plotted points of interest (POIs), such as cafes and gift shops within the same district. Moving forward, we downloaded park polygons and building footprints, visualizing these core urban features. Lastly, we explored how to download road networks of different types within the 1st district of Budapest. This comprehensive approach to geospatial data retrieval and visualization highlights the versatility and power of OSMnx for spatial data science.

Spatial Networks



Analyzing spatial networks, for instance, road networks or infrastructural networks, is a popular area of geospatial sciences, providing insights into urban planning, transportation systems, power grid optimization, and many others. This chapter explores the various methods and techniques for obtaining, transforming, analyzing, and visualizing spatial networks using dedicated Python libraries such as OSMnx, GeoPandas, and NetworkX. As our primary example, we will analyze road networks, which are perfect

examples of spatial networks, where both the nodes as road intersections and the edges as road segments are embedded in a geographical space.

We will start by downloading and converting road networks from OpenStreetMap (OSM) into GeoPandas GeoDataFrames and then transforming these GeoDataFrames back into spatial networks. Following this, we will explore methods for visualizing spatial networks to understand their structure and connectivity. Then, we will conduct network analytical steps that are particularly useful for spatial networks, such as computing and visualizing the shortest paths between nodes and generating walking distance isochrones. Finally, we will obtain and interpret network statistics, calculate centrality measures, and perform community detection to identify clusters within the network.

81. Road Networks in GeoPandas

Our primary example of spatial networks will be road networks, which provide insights into urban planning, transportation systems, and many other use cases of spatial analytics. As a first technical step, we review how to download road networks from OpenStreetMap and store them in GeoDataFrames, which allows for more detailed spatial analysis and visualization using the powerful capabilities of GeoPandas.

We will start by downloading the road network within a specified administrative boundary using `osmnx` and the name of the target area as input. As an example, we will use the area of the 1st district in Budapest and focus on the driving road network. Then, using the `graph_from_polygon` function, we query the road network, and with the help of the `graph_to_gdfs` function, we project the graph object into separate GeoDataFrames for nodes and edges. Finally, we print out some basic information on the network obtained.

In

```
1 # Import osmnx for geospatial data from OpenStreetMap
2 import osmnx as ox
3
4 # Define the polygon for the 1st district of Budapest
5 admin_district = district, Budapest')
6 admin_poly =
7
8 # Download the road network for driving within the admin boundary
9 G =
```

```
10
11 # Print the number of nodes and edges in the road network
12 print("Number of nodes in the original graph:",
13 print("Number of edges in the original graph:",
14 print()
15
16 # Convert the network graph to GeoDataFrames for nodes and edges
17 nodes, edges =
18
19 # Display the first few rows of the nodes GeoDataFrame
20
21
22 # Display the first few rows of the edges GeoDataFrame
23
24
25
26 # Print the features stored in each GeoDataFrame
27 print("Keys in the nodes table:",
28 print("Keys in the edges table:",
29 print()
30
31 # Print the total number of nodes and edges
32 print("Number of nodes in the node table:", len(nodes))
33 print("Number of edges in the node table:", len(edges))
```

Number of nodes in the original graph: 359

Number of edges in the original graph: 706

| | y | x | street_count | highway | geometry |
|----------|-----------|-----------|--------------|---------|--------------------------|
| osmid | | | | | |
| 35500835 | 47.491114 | 19.045623 | 4 | NaN | POINT (5.04552 47.49111) |
| 35523878 | 47.481203 | 19.045614 | 4 | NaN | POINT (5.04561 47.48120) |
| 35523895 | 47.480550 | 19.040555 | 3 | NaN | POINT (5.04055 47.48055) |

| | u | v | key | osmid | oneway | lanes | ref | highway | maxspeed | reversed | length | bridge | geometry | name | junction | tunnel | width | access |
|----------|-------------|---|----------------------------------|-------|--------|-------|-----------|----------|----------|----------|--------|---|--------------------|------|----------|--------|-------|--------|
| 35500835 | 1676318037 | 0 | [464423032, 0, 07050, 000411009] | True | 3 | 3 | primary | [50, 20] | True | 74.230 | yes | LINESTRING (19.04563 47.49111, 19.04579 47.480... | NaN | NaN | NaN | NaN | NaN | |
| | 1676318015 | 0 | [28189966, 212742356, 56066809] | True | 2 | NaN | secondary | [50, 20] | True | 140.044 | NaN | LINESTRING (19.04563 47.49111, 19.04579 47.480... | Székely Gábor utca | NaN | NaN | NaN | NaN | |
| 35523878 | 10921844764 | 0 | [24255570, 56066397] | True | 11, 21 | NaN | secondary | 50 | True | 21.745 | NaN | LINESTRING (19.04561 47.48120, 19.04525 47.481... | Művelődési központ | NaN | NaN | NaN | NaN | |

Keys in the nodes table: ['y', 'x', 'street_count', 'highway', 'geometry']

Keys in the edges table: ['osmid', 'oneway', 'lanes', 'ref', 'highway', 'maxspeed', 'reversed', 'length', 'bridge', 'geometry', 'name', 'junction', 'tunnel', 'width', 'access']

Number of nodes in the node table: 359

Number of edges in the node table: 706

The output of the previous cell provides detailed information about the road network in the 1st district of Budapest. For instance, it shows that the road network specific to driving contains 359 nodes and 706 edges both as a queried graph and when transformed into a node and edge table.

The detailed queries to print the features stored in these tables reveal further information stored in the OSM data, such as geometries, the OSM IDs, or the length of the road segments, allowing us to browse and utilize various types of information describing these spatial networks.

82. From GeoDataFrames to Spatial Networks

After turning a road network graph into GeoDataFrames, we are now doing the reverse operation by transforming GeoDataFrames into spatial networks. This step, complementing the previous one, enables us to leverage graph theory for transportation planning, network optimization, and other use cases. By converting the node and edge tables stored in GeoPandas GeoDataFrames into a [NetworkX](#) graph, we will be able to perform a wide range of network analytics exercises, as we will learn later on in this chapter.

In this section, we will demonstrate how to convert the edges of a road network GeoDataFrame into a networkx graph. As between two intersections, there may be multiple links, as driving lanes, we use the nx.MultiGraph object for efficient data storage. This process involves iterating through the edges GeoDataFrame and adding each edge to the NetworkX graph with appropriate attributes, such as the length of each road segment. We also extract information from the nodes table, as we include the node locations as coordinate pairs to ensure the resulting graph contains the relevant spatial information.

In

```
1 # Import the libraries we use NetworkX for network analysis
2 import networkx as nx
3 import osmnx as ox
4
5
```

```
6 # Define the polygon for the admin boundary the 1st district of
Budapest

7 admin_district = ox.geocode_to_gdf('1st district, Budapest')
8 admin_poly = admin_district.geometry.values[0]
9
10 # Download the road network for driving within the administrative
boundary
11 G = ox.graph_from_polygon(admin_poly, network_type='drive')
12
13 # Print the number of nodes and edges in the road network
14 print("Number of nodes in the original graph:", G.number_of_nodes())
15 print("Number of edges in the original graph:", G.number_of_edges())
```

Number of nodes in the original graph: 359

Number of edges in the original graph: 706

In

```
1 # Convert the network graph to GeoDataFrames for nodes and edges
2 nodes, edges = ox.graph_to_gdfs(G)
3
4 # Print the number of nodes and edges in the road network
5 print("Number of nodes in the node table:", len(nodes))
6 print("Number of edges in the edge table:", len(edges))
```

Number of nodes in the node table: 359

Number of edges in the edge table: 706

In

```
1 # Create an empty graph
2 G = nx.MultiGraph()
3
4 # Add edges to the graph
5 for idx, row in edges.iterrows():
6     G.add_edge(idx[0], idx[1], weight=row['length'])
7
8 # Add nodes with location data to the graph
9 for idx, row in nodes.iterrows():
10     G.add_node(idx, y=row['y'], x=row['x'])
11
12 # Print the number of nodes and edges in the graph
13 print("Number of nodes in the reconstructed graph:",
14       G.number_of_nodes())
15 print("Number of edges in the reconstructed graph:",
16       G.number_of_edges())
```

Number of nodes in the reconstructed graph: 359

Number of edges in the reconstructed graph: 706

In

```
1 # Convert node attributes to a list
2 nodes_dict =
3 print(nodes_dict[0:5])
4 print()
```



```
5
6 # Convert edge attributes to a list
7 edges_dict =
8 print(edges_dict[0:5])
```

```
[(35500835, {'y': 47.4901141, 'x': 19.0456276}), (1676318037, {'y':
47.4899404, 'x': 19.046544}), (1676318015, {'y': 47.4892281, 'x':
19.0467569}), (35523878, {'y': 47.4913026, 'x': 19.0456141}),
(10921844764, {'y': 47.4912206, 'x': 19.0452095})]
```

```
[(35500835, 1676318037, {'weight': 74.333}), (35500835, 1676318015,
{'weight': 132.01100000000002}), (35500835, 1676318029, {'weight':
723.6199999999999}), (35500835, 1676318118, {'weight': 71.951}),
(1676318037, 1676318015, {'weight': 100.84100000000002})]
```

In this section, we overviewed how we can create a spatial network by turning node and edge GeoDataFrame tables, initially exported from a road network graph, back into the road network graph format while preserving all nodes and links in the graph. Besides tracking the size of the network throughout the processing steps, we also confirmed that all attributes are added to the spatial network by printing the lists of nodes and edges.

83. Visualizing Spatial Networks

Visualizing spatial networks is an essential step in geospatial analysis, allowing us to manually observe and interpret the structure, connectivity, and spatial distribution of network elements such as roads and intersections. By creating visual representations of spatial networks, we can gain valuable insights into various use cases, set up model hypotheses, and select modeling and analytical approaches accordingly.

In this section, we will demonstrate how to visualize a spatial network. First, we will use the built-in visualization routine of `osmnx` and then the more customizable plotting toolkit of `matplotlib` setting specific plotting parameters, such as the attribute used to color the edges the colormap or the line style. On both visuals, we will create visual representations of the network graph, highlighting nodes and edges to showcase the overall structure and connectivity of the network of the 1st district of Budapest. As in the previous examples, the code snippet below is easily adaptable to any other place name or polygon geometry.

In

```
1 # Import osmnx and matplotlib
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4
5 # Define the polygon for the admin boundary the 1st district of Budapest
6 admin_district = ox.geocode_to_gdf('1st district, Budapest')
7 admin_poly = admin_district.geometry.values[0]
8
```

```
9 # Download the road network for driving within the administrative
boundary
10 G = ox.graph_from_polygon(admin_poly, network_type='drive')
11
12 # Plot the network using OSMnx
13 fig, ax = ox.plot_graph(G,
14                         node_size=10,
15                         node_color='blue',
16                         edge_color='gray',
17                         bgcolor='white',
18
19
20
21 # Show the road network
22 ax.set_title('Spatial Network Visualization with OSMnx', fontsize=16)
23 plt.show()
```

Spatial Network Visualization with OSMnx



In

```
1 # Create a plot to visualize the spatial network
2 f, ax = plt.subplots(1, 1, figsize=(8, 8))
3
4 # Convert the network graph to GeoDataFrames for nodes and edges

5 nodes, edges = ox.graph_to_gdfs(G)
6
7 # Plot the edges and nodes, coloring edges based on their length
8 edges.plot(column = 'length', ax=ax, linewidth=2, cmap = 'Reds')
9 nodes.plot(ax=ax, markersize=15, alpha = 0.9, color='crimson')
10
11 # Customize the plot
12 ax.set_title('Spatial Network Visualization with Matplotlib', fontsize=16)
13 ax.set_axis_off()
14 plt.show()
```

Spatial Network Visualization with Matplotlib



Visualizing spatial networks using `osmnx` and `matplotlib` offers a clear and detailed depiction of the network's structure and connectivity. For example, we can observe a concentric pattern of lines representing the Buda Hill and

the Castle District. In the second figure, the varying lengths of drivable road segments are highlighted by different colors. Such maps and road network structures are invaluable for urban planners and transportation engineers, providing insights into the local context and constraints defined by the city's inherent structure.

84. Calculating Shortest Paths

Calculating the shortest path in a spatial network is a fundamental task in areas like transportation and route planning or navigation. By identifying the most efficient route between two points, we can optimize travel times, reduce congestion, and improve overall network efficiency. Using the previously introduced NetworkX library designed for graph analytics, we can incorporate graph-based algorithms to compute the shortest path within a spatial network.

In this section, we will demonstrate how to calculate the shortest path between two randomly chosen nodes in the example road network from the 1st district of Budapest. We will use the `shortest_path` function from NetworkX, specifying the road segment lengths as weights to find the most efficient route and output the shortest path expressed by a list of connected nodes, each denoted by its OSM ID.

In

```
1 # Necessary library imports
2 import networkx as nx
3 import random
4
5
6 # Define the polygon for the 1st district of Budapest
7 admin_district = ox.geocode_to_gdf('1st district, Budapest')
8 admin_poly = admin_district.geometry.values[0]
9
```



```
10 # Download the road network for driving within the admin boundary

11 G = ox.graph_from_polygon(admin_poly, network_type='drive')
12
13 # Convert the GeoDataFrame nodes to a list of unique node indices
14 nodes_l = list(set(nodes.index))
15
16 # Define the origin and destination nodes
17 origin = random.choice(nodes_l)
18 destination = random.choice(nodes_l)
19 print("Randomly picked origin intersection: ", origin)
20 print("Randomly picked destination intersection: ", destination)
21
22 # Calculate the shortest path
23 shortest_path = nx.shortest_path(G,
24                                   source=origin,
25                                   target=destination,
26                                   weight='weight')
27 print("\nShortest path:", shortest_path)
28
29 # Calculate the total length of the shortest path
30 total_length = 0
31 for i in range(len(shortest_path) - 1):
32     total_length += G[shortest_path[i]][shortest_path[i + 1]][0]['length']
33
34 print("\nTotal length of the shortest path:", len(shortest_path),
35       "nodes")

35 print("Total length of the shortest path:", total_length, "meters")
```

Randomly picked origin intersection: 223783216

Randomly picked destination intersection: 4886199474

Shortest path: [223783216, 1676318007, 1676318029, 1676318032, 1676317984, 1676317982, 277367774, 436615230, 277367628, 277367636, 265597672, 265597674, 277888713, 277888709, 4886199474]

Total length of the shortest path: 15 nodes

Total length of the shortest path: 2079.821 meters

The results of this computation show the OSM IDs of the two randomly picked nodes, the shortest path as a series of connected nodes denoted by their OSM IDs, the total number of graph hops, and the total distance computed from the edge length attributes.

85. Visualizing the Shortest Path

In this section, we learn how to visualize the shortest path connecting two random intersections of a road network. In general, by highlighting the shortest path on a map, urban planners and transportation engineers can gain valuable insights into the network's performance and identify areas for improvement.

Using GeoPandas and Matplotlib, we can visualize the shortest path along the entire network, as demonstrated in our usual example district from Budapest. Here, we first pick two random nodes and then compute the shortest path between them using of the NetworkX library. Then, we define a function which turns the shortest path, a list of connected nodes, into a list of connected edges. Finally, we plot the entire network and highlight the edges belonging to the shortest path in red.

In

```
1 # Necessary library imports
2 import matplotlib.pyplot as plt
3 import random
4 import networkx as nx
5
6 # Define the polygon for the 1st district of Budapest
7 admin_district = ox.geocode_to_gdf('1st district, Budapest')
8 admin_poly = admin_district.geometry.values[0]
9
10 # Download the road network for driving within the administrative
    boundary
```

```
11 G = ox.graph_from_polygon(admin_poly, network_type='drive')
12
13 # Convert the GeoDataFrame nodes to a list of unique node indices
14 nodes_l = list(set(nodes.index))
15
16 # Define the origin and destination nodes
17 origin = random.choice(nodes_l)
18 destination = random.choice(nodes_l)
19
20 # Calculate the shortest path
21 shortest_path_nodes = nx.shortest_path(G,
22                                     source=origin,
23                                     target=destination,
24                                     weight='weight')
25
26 print("The first three nodes of the shortest path: ",
shortest_path_nodes[0:3])
```

The first three nodes of the shortest path: [1676318029, 1676318032, 1676317984]

In

```
1 def get_path_edges(G, path):
2     edges = []
3     for i in range(len(path) - 1):
4         edges.append((path[i], path[i + 1], 0))
5     return edges
6
```

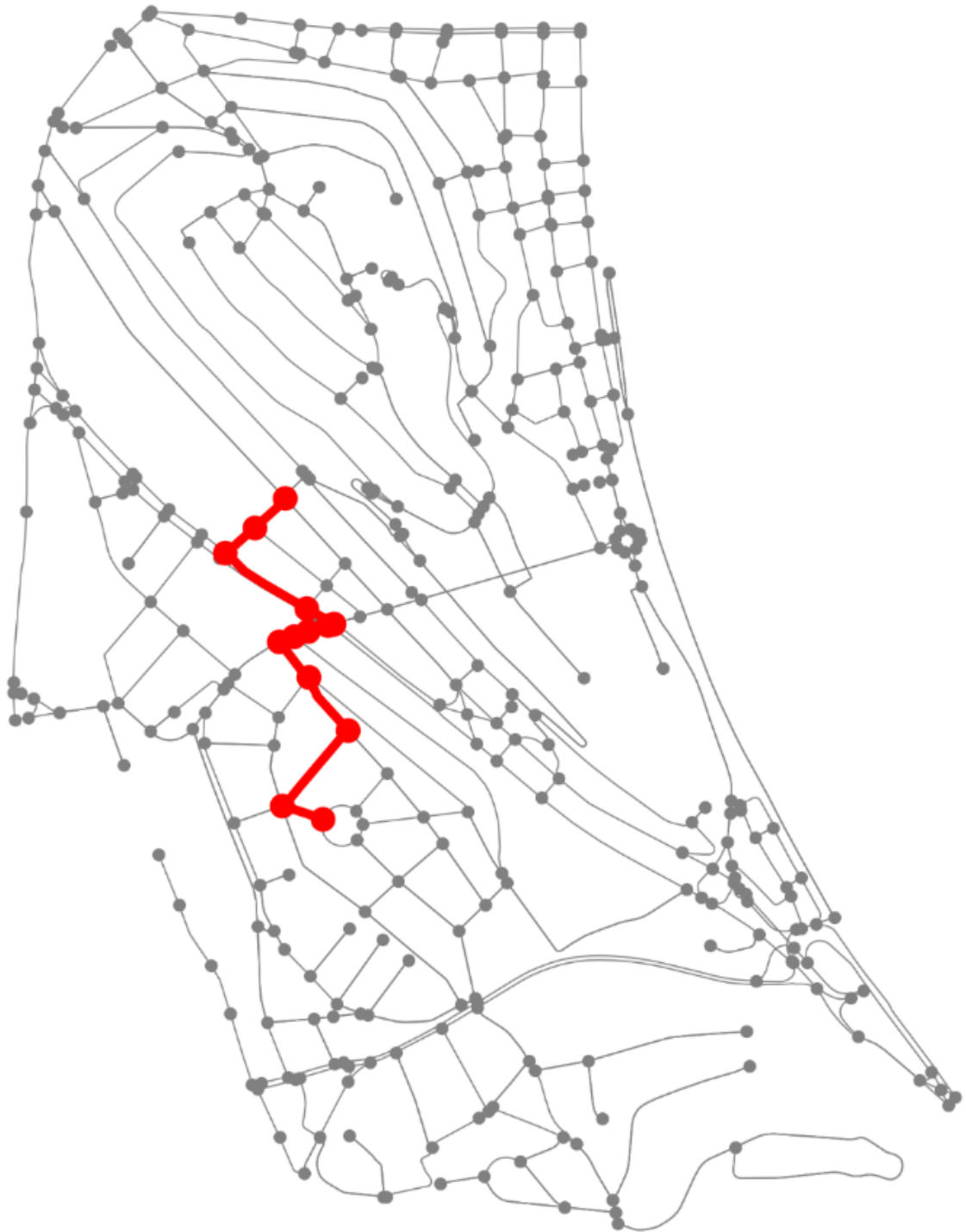
```
7 shortest_path_edges = get_path_edges(G, shortest_path_nodes)
8 print("The first three edges of the shortest path: ",
shortest_path_edges[0:3])
```

The first three edges of the shortest path: [(1676318029, 1676318032, 0),
(1676318032, 1676317984, 0), (1676317984, 1676317982, 0)]

In

```
1 # Plot the entire network and the shortest path
2 f, ax = plt.subplots(1, 1, figsize=(8, 8))
3
4 # Plot all edges and nodes
5 edges.plot(ax=ax, linewidth=0.5, color='grey')
6 nodes.plot(ax=ax, markersize=10, color='grey')
7
8 # Plot the shortest path edges in red
9 shortest_path_edges_gdf = edges.loc[shortest_path_edges]
10 shortest_path_edges_gdf.plot(ax=ax, linewidth=3, edgecolor='red')
11
12 # Plot the shortest path nodes in red
13 shortest_path_nodes_gdf = nodes.loc[shortest_path_nodes]
14 shortest_path_nodes_gdf.plot(ax=ax, markersize=50, color='red',
zorder=5)
15
16 ax.set_title('Shortest Path Visualization', fontsize=16)
17 ax.set_axis_off()
18 plt.show()
```

Shortest Path Visualization



Here, we visualized the shortest path between two random points in the driving road network of District I in Budapest, giving an example of how to conduct efficient route planning on a spatial network. By integrating these visualization techniques, we can better understand route optimization, transportation planning, and various other network processes in urban settings.

86. Generating Walking Distance Isochrones

In geospatial sciences, isochrones represent areas that can be reached within a certain amount of time from a specific point. In the context of walking distances, isochrones help urban planners and transportation engineers understand accessibility and connectivity within an urban area. By visualizing isochrones, we can gain insights into the walkability of neighborhoods and identify areas that are well-connected or, for instance, in need of improved pedestrian infrastructure.

In this section, we will demonstrate how to generate walking distance isochrones using `osmnx` and `First`. First, we pick a random road network node in the network of Budapest's 1st district as a starting point. Second, assuming a constant walking speed of 5km/h, we compute the travel time needed to traverse each edge of the road network. Then, we use Polygon-based calculations to determine the edges of the areas that can be reached from the center point within 1, 2, 5, and 10 minutes of walking. Finally, we visualize the isochrones as overlaying polygons using `matplotlib` and coloring them from green to red as the reaching time increases.

In

```
1 # Necessary library imports
2 import osmnx as ox
3 import networkx as nx
4 import matplotlib.pyplot as plt
5 import geopandas as gpd
6 from shapely.geometry import Point, Polygon
7 import matplotlib.patches as mpatches
```


8

9

```
10 # Define the polygon for the 1st district of Budapest
11 admin_district = ox.geocode_to_gdf('1st district, Budapest')
12 admin_poly = admin_district.geometry.values[0]
13
14 # Load the graph from OSMnx
15 G = ox.graph_from_polygon(admin_poly, network_type='walk')
16
17 # Define the walking speed (5 km/h -> 1.39 m/s)
18 walking_speed = 1.39 # in meters per second
```

In

```
1 # Calculate travel time for each edge
2 for u, v, data in
3     # Calculate travel time in seconds
4     data['travel_time'] = data['length'] / walking_speed
5
6 # Generate isochrones
7 center_node = list(G.nodes())[30] # starting point
8 isochrone_times = [1, 2, 5, 10] # isochrones in minutes
9 isochrone_polys = []
10
11 for time in isochrone_times:
12     subgraph = nx.ego_graph(G,
13                             center_node,
14                             radius=time*60,
15                             distance='travel_time')
```

16

```
17     node_points = [Point((data['x'], data['y'])) \
18                     for node, data in
19
20     polygon =
21     Polygon(gpd.GeoSeries(node_points).unary_union.convex_hull)
22     isochrone_polys.append(gpd.GeoSeries([polygon]))
```

In

```
1 # Create visuals
2 fig, ax = plt.subplots(figsize=(8, 8))
3
4 # Define the colors of the isochrones
5 colors = ['red', 'orange', 'yellow', 'green']
6
7 # Plot the isochrones
8 for idx, (polygon, time) in \
9     enumerate(reversed(list(zip(isochrone_polys, isochrone_times)))):
10     polygon.plot(ax=ax, color=colors[idx], alpha=0.9, label=f'{time} min')
11
12 # Manually create legend handles
13 handles = [
14     mpatches.Patch(color=colors[idx], alpha=0.9, label=f'{time} min')
15     for idx, time in enumerate(reversed(isochrone_times))
16 ]
17
18 # Plot the network
19 ox.plot_graph(G, ax=ax, node_size=0, edge_linewidth=0.5,
```

20

21 # Add the legend to the plot and set the title

22 ax.set_title('Isochrones', fontsize=16)

23 legend = plt.legend(handles=handles)

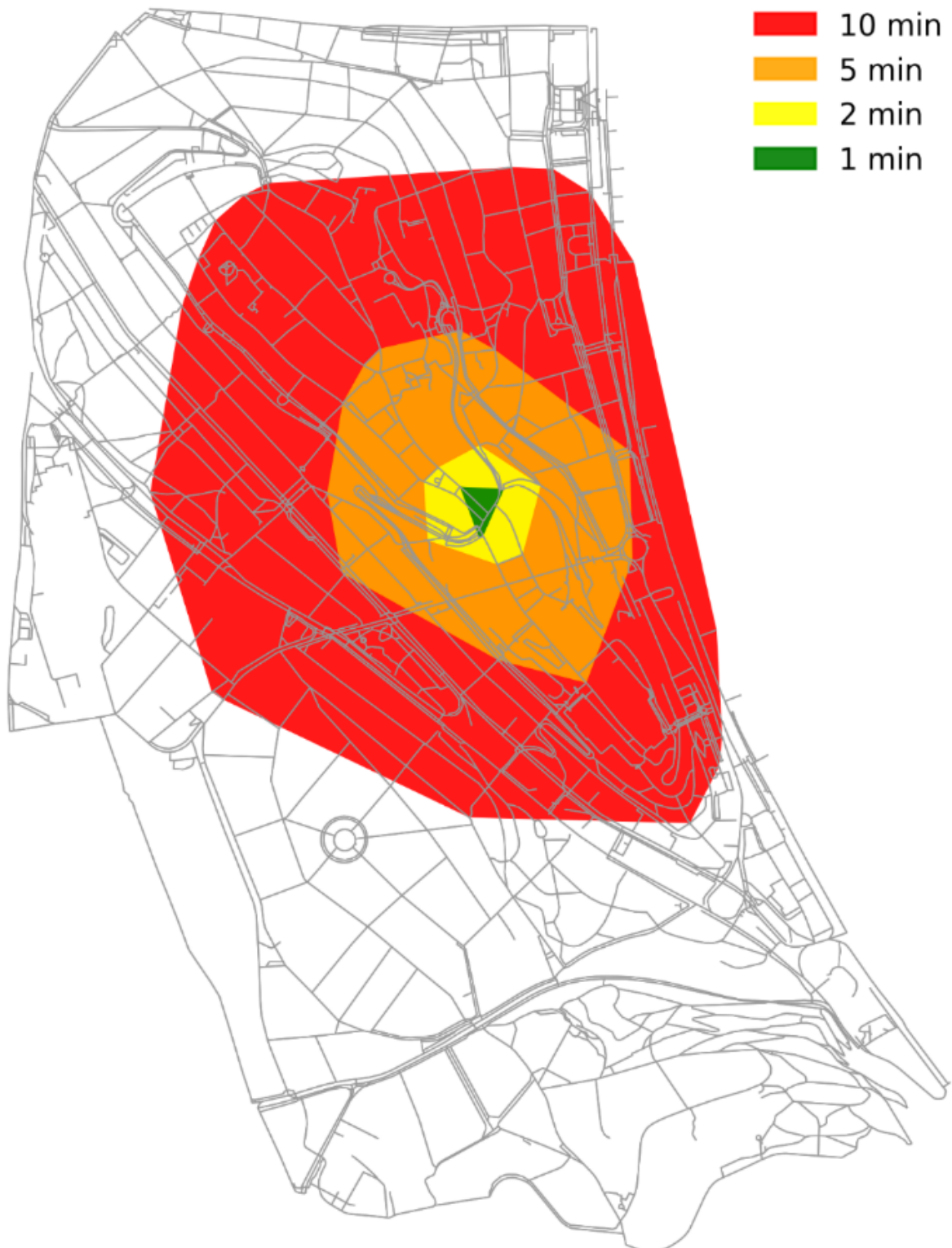
24 legend.get_frame().set_linewidth(0) # Remove the legend box's frame

25

26 # Show the plot

27 plt.show()

Isochrones



The isochrone figure shows how the different isochrone layers grow proportionally to the increased time frame, representing reachable areas

within 1, 2, 5, and 10 minutes of walking from the origin.

87. Obtaining Spatial Network Statistics

Analyzing the statistical properties of a spatial network is essential for understanding its structure, efficiency, and overall connectivity. Network statistics provide valuable insights into various aspects of the network, such as the number of nodes and edges, network density, average node degree, and more. Using the built-in stats tool of the osmnx library, we can easily compute these basic descriptive statistics for our example spatial network.

In

```
1 # Import osmnx for geospatial data from OpenStreetMap
2 import osmnx as ox
3
4 # Define the polygon for the admin boundary of the 1st district
5 admin_district = ox.geocode_to_gdf('1st district, Budapest')
6 admin_poly = admin_district.geometry.values[0]
7
8 # Load the graph from OSMnx
9 G = ox.graph_from_polygon(admin_poly, network_type='drive')
10
11 # Obtain basic statistics for the spatial network
12 basic_stats = ox.stats.basic_stats(G)
13 basic_stats
```

Out

```
{'n': 359,  
  
'm': 706,  
'k_avg': 3.933147632311978,  
'edge_length_total': 79922.763999999994,  
'edge_length_avg': 113.20504815864014,  
'streets_per_node_avg': 3.01949860724234,  
'streets_per_node_counts': {0: 0, 1: 28, 2: 12, 3: 249, 4: 65, 5: 5},  
'streets_per_node_proportions': {0: 0.0,  
  1: 0.07799442896935933,  
  2: 0.033426183844011144,  
  3: 0.6935933147632312,  
  4: 0.181058495821727,  
  5: 0.013927576601671309},  
'intersection_count': 331,  
'street_length_total': 56695.9319999999986,  
'street_segment_count': 523,  
'street_length_avg': 108.405223709369,  
'circuitry_avg': 1.0571277079153225,  
'self_loop_proportion': 0.0019120458891013384}
```

The above simple statistical overview of the example road network provides a wide range of insights about the network's structure and connectivity. For instance, with 359 nodes and 706 edges, the network has a relatively high average node degree of 3.93, indicating good connectivity. The average edge length of 113.35 meters and the total edge length of 80,023 meters provide information on the network's scale and typical road segment length.

88. Network Centrality Measures

Centrality measures are mathematical tools in graph theory and network analysis designed to identify the most important nodes within a network from various perspectives. These measures both capture insights about a network's overall structure and node significance, such as connectivity, influence, and accessibility. By analyzing centrality measures, domain experts can pinpoint key intersections and routes that are vital for efficient network functioning, as well as identify points where external interventions are needed to improve the networks.

In this section, we will first create the usual sample spatial network about the first district of Budapest. Then, we calculate three common centrality measures: degree centrality, betweenness centrality, and closeness centrality, which we further explain after this paragraph. While we compute the network-level average values of each of these metrics, in the final step, we build a DataFrame containing the full centrality profile of each node as well.

Degree This measure quantifies the number of connections a node has. In a geospatial context, it represents the number of roads connected to an intersection, highlighting how well-connected a particular node is within the network.

Betweenness This measure captures the extent to which a node lies on the shortest paths between other nodes. In geospatial analysis, it identifies key intersections that act as critical bridges and bottlenecks within the

network, suggesting points that might control traffic flow or have significant influence over the connectivity of the area.

Closeness This measure reflects how close a node is to all other nodes in the network. In the context of geospatial networks, it indicates how quickly one can reach all other intersections from a given node, emphasizing nodes that are centrally located and easily accessible.

In

```
1 # Library imports
2 import networkx as nx
3 import osmnx as ox
4 import pandas as pd
5
6 # Define the polygon for the 1st district of Budapest
7 admin_district = ox.geocode_to_gdf('1st district, Budapest')
8 admin_poly = admin_district.geometry.values[0]
9
10 # Load the graph from OSMnx
11 G = ox.graph_from_polygon(admin_poly, network_type='walk')
```

In

```
1 # Degree centrality
2 degree_centrality = nx.degree_centrality(G)
3
4 # Betweenness centrality
```

```

5 betweenness centrality = nx.betweenness centrality(G,
weight='length')
6
7 # Closeness centrality
8 closeness centrality = nx.closeness centrality(G)
9
10 # Calculate the number of nodes
11 num_nodes = G.number_of_nodes()
12
13 # Print the average values for each centrality measure
14 degree centrality:
15     {sum(degree centrality.values()) / num_nodes:.4f}")
16 betweenness centrality:
17     {sum(betweenness centrality.values()) / num_nodes:.4f}")
18 closeness centrality:
19     {sum(closeness centrality.values()) / num_nodes:.4f}")

```

In

```

1 # Create DataFrames for each centrality measure
2 df_degree centrality =
3     'degree centrality'
4     ]).set_index('osmid')
5 df_betweenness centrality =
6     'betweenness centrality'
7     ]).set_index('osmid')
8 df_closeness centrality =
9     'closeness centrality'

```

```
10         ]).set_index('osmid')
11
12 # Merge the DataFrames into a single DataFrame
13 df_centralities =
df_degree_centrality.merge(df_betweenness_centrality,
14
15
16 df_centralities = df_centralities.merge(df_closeness_centrality,
17
18
19
20 # Display the first few rows of the DataFrame
21 df_centralities.head()
```

In this section, we computed three different network centralities on the level of the selected example road network. While we attached unique centrality values to each node stored in a DataFrame, we also computed network-level averages.

89. Community Detection in Spatial Networks

Community detection is one of the foundational analytical steps in network analysis and aims to identify groups of nodes that are more densely connected to each other than to the rest of the network. In geospatial networks, such as street networks, detecting communities can reveal clusters of intersections and streets that form natural subdivisions within the city. This information is valuable for urban planners, transportation engineers, and city officials to understand the functional organization of urban areas.

In this section, we will demonstrate how to perform community detection in a spatial network using the [greedy modularity optimization algorithm](#) implemented in We will then visualize the network with nodes colored according to their community membership.

In

```
1 # Necessary imports
2 import osmnx as ox
3 from networkx.algorithms.community import
  greedy_modularity_communities
4
5 # Define the polygon for the 1st district of Budapest
6 admin_district = ox.geocode_to_gdf('1st district, Budapest')
7 admin_poly = admin_district.geometry.values[0]
8
9 # Load the graph from OSMnx
```

```
10 G = ox.graph_from_polygon(admin_poly, network_type='walk')

11
12 # Perform community detection
13 communities = greedy_modularity_communities(G)
14
15 # Create a dictionary mapping nodes to their community
16 node_community = {}
17 for i, community in enumerate(communities):
18     for node in community:
19         node_community[node] = i
20
21 # Add community data to the nodes GeoDataFrame
22 nodes['community'] = nodes.index.map(node_community)
```

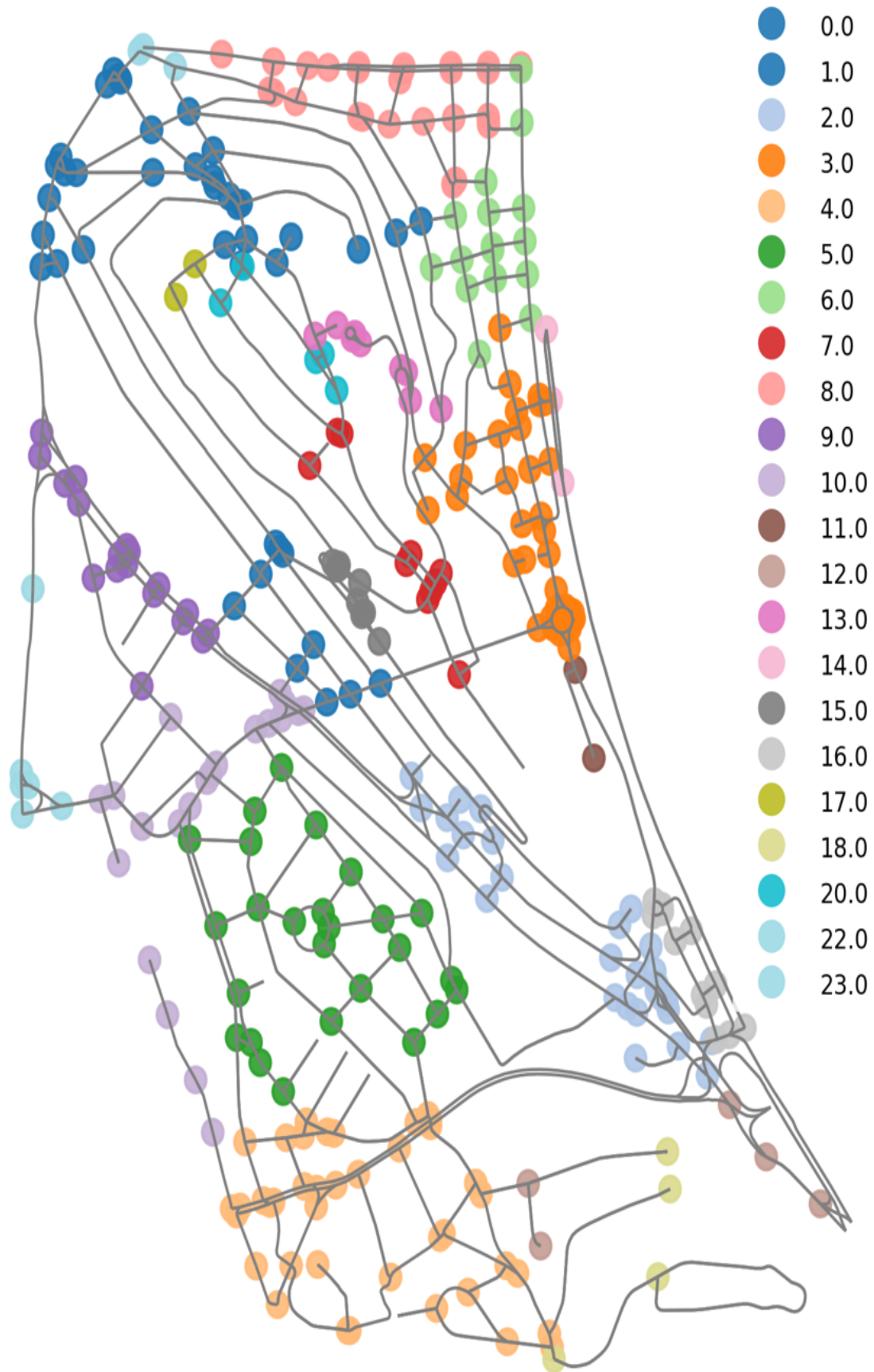
In

```
1 # Plot the network with nodes colored by community
2 fig, ax = plt.subplots(figsize=(8, 8))
3 nodes_plot = nodes.plot(ax=ax,
4                           column='community',
5
6
7                           markersize=60,
8                           alpha=0.9,
9                           cmap='tab20')
10
11 edges.plot(ax=ax, color='gray', linewidth=1)
12
13 # Set the title
```

```
14 ax.set_title('Community Detection in Street Network', fontsize=16)

15 ax.axis('off')
16
17 # Change the font size of the legend and remove the frame
18 legend = ax.get_legend()
19 if legend is not
20     plt.setp(legend.get_texts(), fontsize=8)
21     legend.get_frame().set_linewidth(0)
22
23 plt.show()
```

Community Detection in Street Network

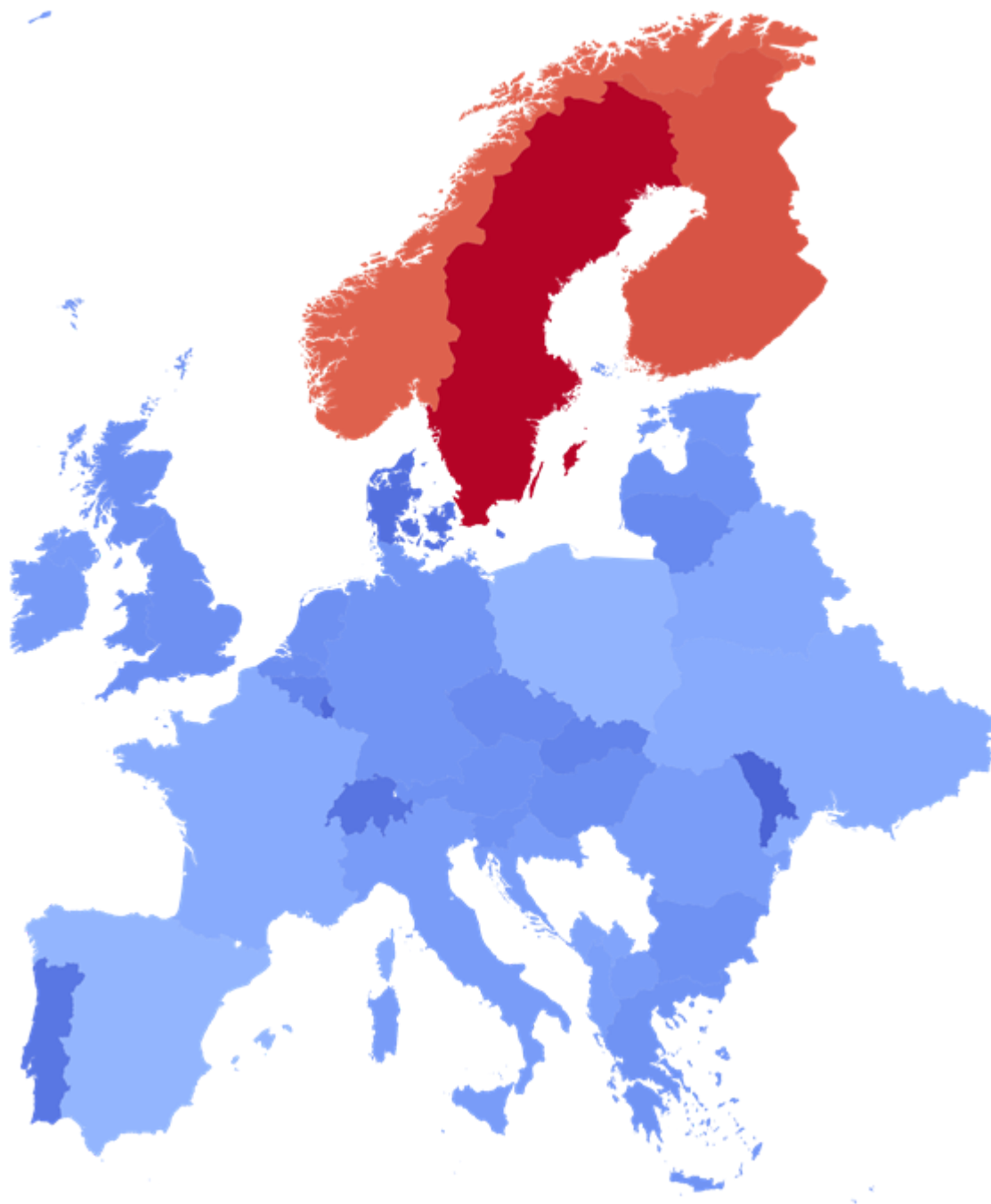


This figure shows each node colored based on its network community - a densely interconnected subset of nodes. These groups usually form stronger, more coherent subnetworks within the whole network, corresponding to various highly interlinked, more homogenous urban forms in a city's road network.

Summary on Spatial Networks

In this chapter, we covered a comprehensive range of techniques for analyzing and visualizing spatial networks. We began by converting road networks from OSM into GeoPandas GeoDataFrames, then transforming these GeoDataFrames back into spatial networks. We demonstrated how to calculate and visualize shortest paths, generate walking distance isochrones, and obtain network statistics. Additionally, we explored centrality measures to identify key intersections and routes and performed community detection to uncover natural subdivisions within the network. Overall, we reviewed several network science and graph analytics tools and methods that are capable of supporting the daily work of spatial data scientists, urban planners, and GIS analysts alike.

Geospatial Statistics and Machine Learning



In this final chapter, we overview several tools and techniques from the topics of spatial statistics, machine learning models, and spatial modeling,

providing practical examples using a handful of relevant Python libraries.

We begin with descriptive statistics, using GeoPandas to derive basic statistical measures and visualize the distributions of geospatial attributes. Next, we explore global and local spatial autocorrelation, using Moran's I for global spatial patterns and Local Moran's I for local clustering and dispersion patterns.

Then, we move on to cover spatial feature engineering, demonstrating how to create new attributes based on spatial relationships. Building on these features, we present various regression models, such as Ordinary Least Squares (OLS) regression and Spatial Lag Model (SLM), and present a framework to compare their goodness-of-fit metrics and predictive power.

Next, we dive into spatial machine learning tools, such as hotspot analysis using the Getis-Ord G_i^* statistic and Kernel Density Estimation (KDE) for point pattern analysis. We conclude the chapter with interpolation methods like Inverse Distance Weighting (IDW) and spatial clustering using DBSCAN (Density-Based Spatial Clustering of Applications with Noise) and K-Means algorithms.

By the end of this chapter, we will have a comprehensive understanding of how to leverage various statistical tools, spatial modeling techniques, and machine learning models to analyze and predict spatial patterns effectively.

Preparing the Maps of Europe and Africa

Throughout the following sections, we will use the maps of Europe and Africa derived from the global map within the Natural Earth data set. Additionally, we will compute the area and length of each polygon enclosing the selected countries. To make sure we don't repeat the same steps over and over, we start the chapter by preparing these data sets.

In

```
1 # Library import
2 import geopandas as gpd
3 from shapely.geometry import box
4 import matplotlib.pyplot as plt
5
6 # Define bounding box for Europe
7 bbox =
8 bbox =
9
10 # Load world countries dataset
11 gdf =
12
13 # Calculate area and length of each geometry
14 gdf['area'] =
15 gdf['length'] =
16
17 # Filter to Europe and exclude Russia
18 gdf_europe = gdf[gdf['continent'] == 'Europe']
19 gdf_europe = gdf_europe[gdf_europe['country'] != 'Russia']
```

```
20 gdf_europe = != 'Russia']
```

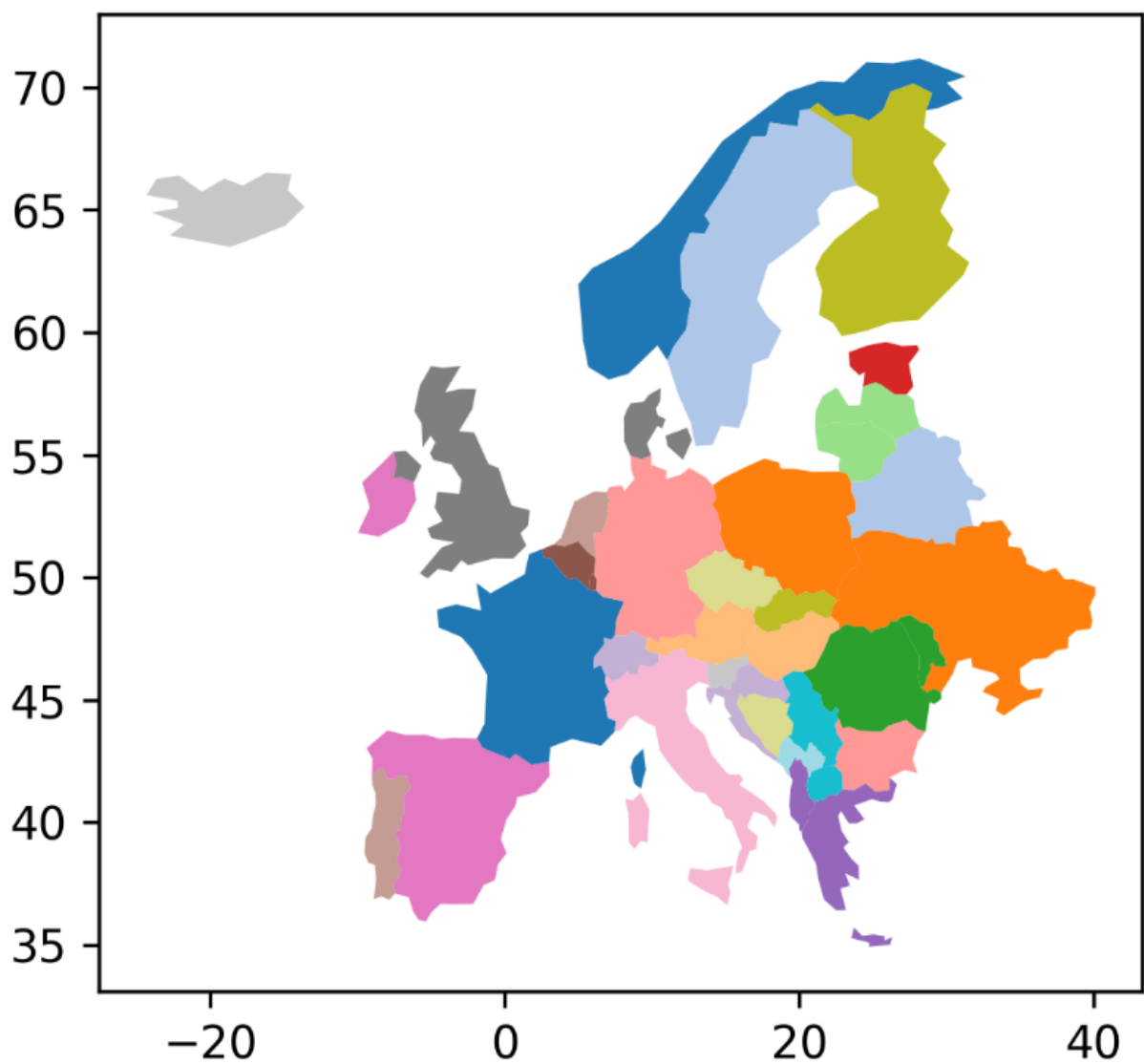
```
21
```

```
22 # Plot Europe dataset
```

```
23 = 'tab20')
```

Out

```
>
```



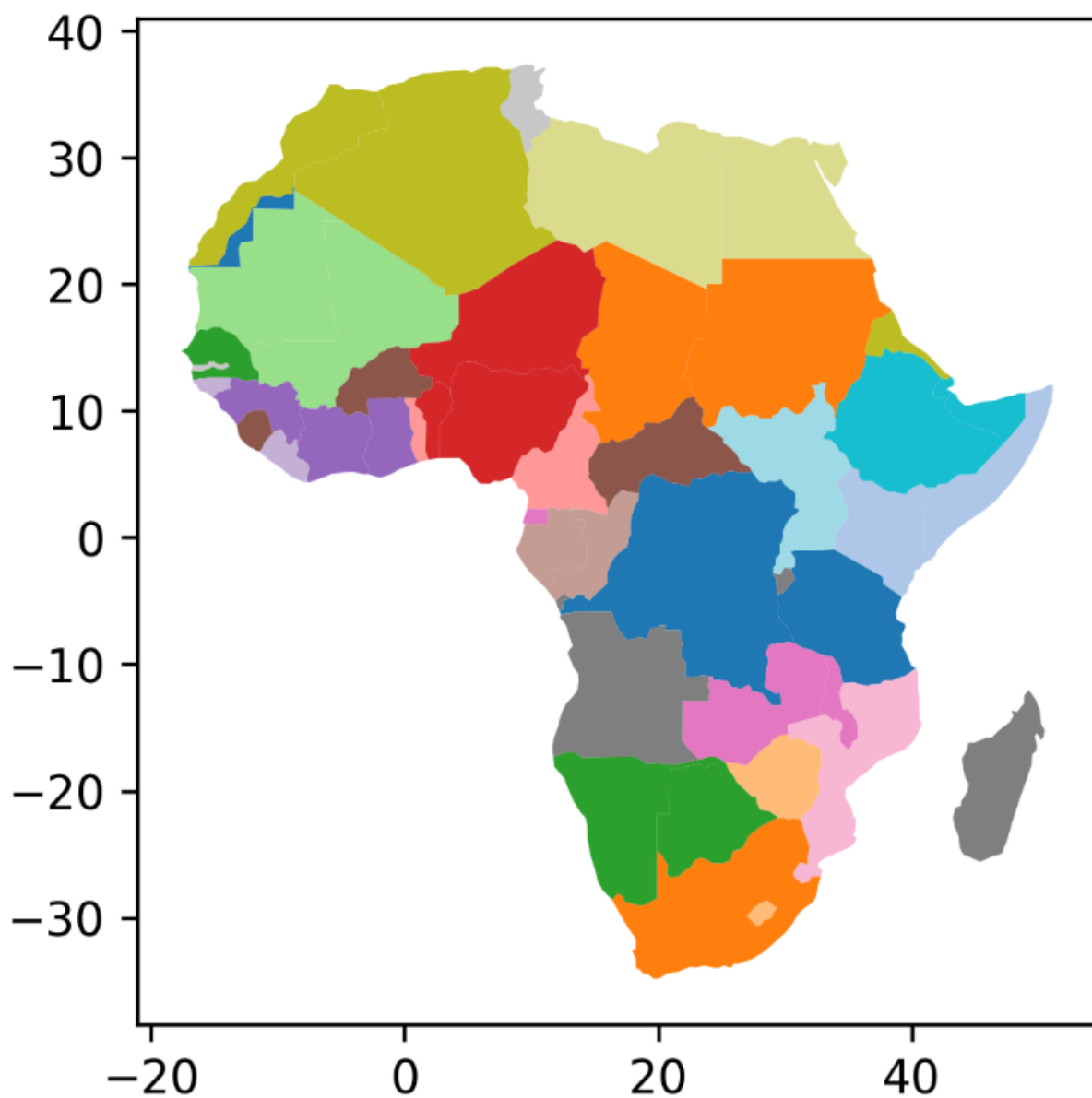
In

```
1 # Filter for African countries
2 gdf_africa = gdf[gdf['continent'] == 'Africa']
3
4 # Calculate area and length of each geometry
5 gdf_africa['area'] =
6 gdf_africa['length'] =

7
8 = 'tab20')
```

Out

>



90. Descriptive Statistics with GeoPandas

Deriving descriptive statistics on geospatial data offers the first quantitative glimpse into the hidden patterns and relationships behind any set of information, setting the stage for more advanced statistical and machine learning applications and data exploration.

In this section, we will use the previously prepared map of Europe. Then, we compute the descriptive statistics on its continuous features, the area and length of each country's boundary polygon as well as the built-in population estimates and GDP estimates of each country.

We use the `GeoDataFrame.describe()` method to generate basic descriptive statistics for the numeric columns, displaying key measures such as mean, standard deviation, and quartiles. Then, we display the largest and smallest, poorest, and richest countries by sorting and displaying the `GeoDataFrame` accordingly. To further dive into the characteristics of these features, we also create histograms using `matplotlib.pyplot` to show the distributions of the four continuous features.

In

```
1 # Library import
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4
5 # Show the largest and smaller countries
6 print("The largest and smallest countries:")
7 = 'area',
```

```
8         ascending = "area"]])
```

```
9 = 'area',
```

```
10         ascending = "area"]])
```

The largest and smallest countries:

| | | | | | | | |
|------------|------------|------------|------------|------------|------------|------------|------------|
| countries: | countries: | countries: | countries: | countries: | countries: | countries: | countries: |
| | | | | | | | |

In

```
1 # Show the most and least populated countries
2 print("The most and least populated countries:")
3 display(gdf_europe.sort_values(by = 'pop_est',
4         ascending = True).head(3)[["name", "pop_est"]])
5 display(gdf_europe.sort_values(by = 'pop_est',
6         ascending = False).head(3)[["name", "pop_est"]])
```

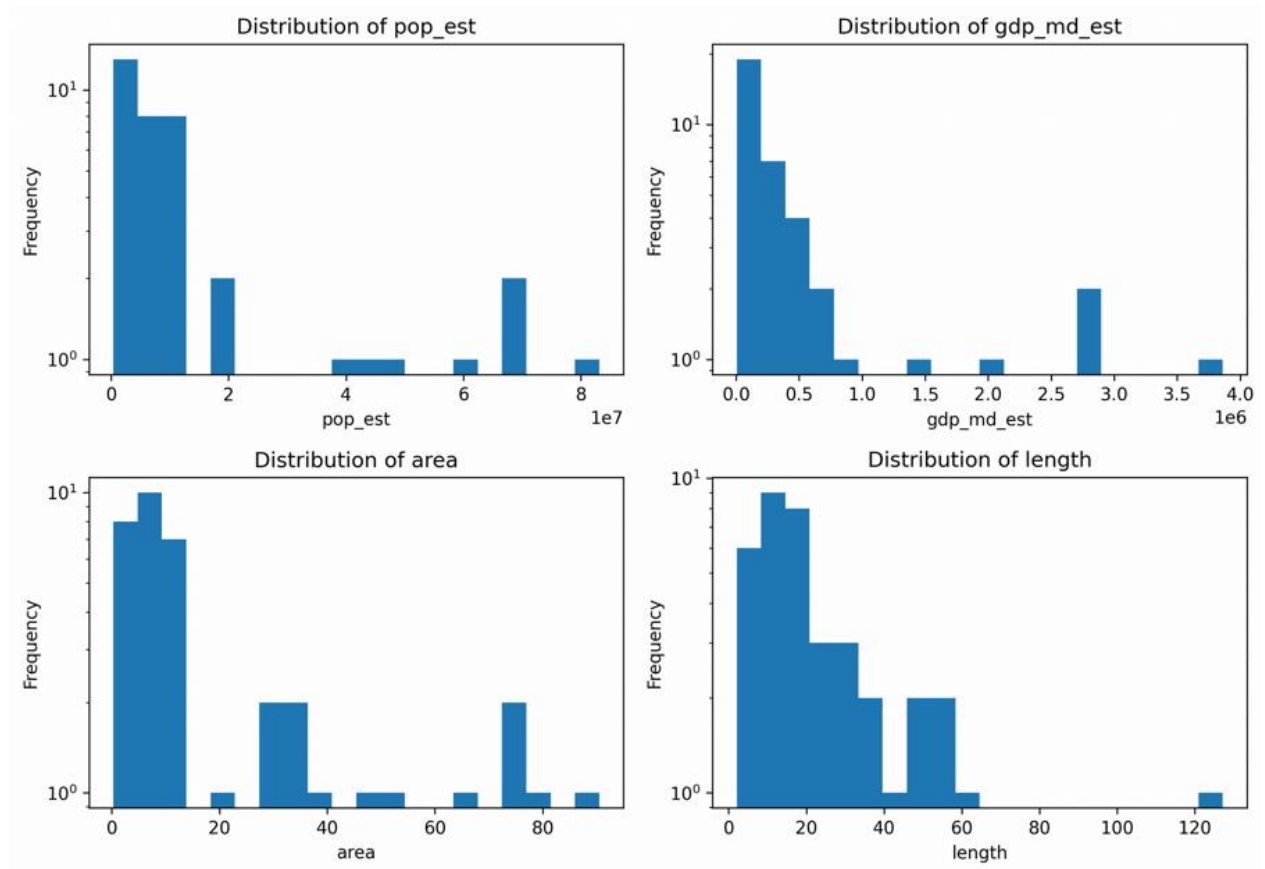
The most and least populated countries:

| | | | | | | | |
|------------|------------|------------|------------|------------|------------|------------|------------|
| countries: | countries: | countries: | countries: | countries: | countries: | countries: | countries: |
| | | | | | | | |

In

```
1 # Show the poorest and richest countries
```

```
4 # Features to plot
5 features = ['pop_est', 'gdp_md_est', 'area', 'length']
6 indices = [(i, j) for i in range(2) for j in range(2)]
7
8 # Plot histograms for each feature
9 for idx, feature in enumerate(features):
10     bx = ax[indices[idx]]
11
12     of' + feature)
13
14
15
16 # Adjust layout for better visualization
17
18
```



The previous code cell outputs, including the histograms, show that both the population estimates and GDP estimates are highly skewed towards lower values, with a few countries having significantly higher values, indicating substantial disparities among European countries. Similarly, the area and length distributions show that most countries are small in size and border length, with a few exceptions contributing to a larger spread in the dataset.

91. Global Spatial Autocorrelation

Global spatial autocorrelation is a widely used analytical concept that measures the degree to which a set of spatial features and their associated data values tend to be clustered together in space. In other words, spatial autocorrelation captures the correlation of a variable with itself through geographies. A popular way of quantifying this is to use Moran's I statistic. Similarly to regular correlation, positive values of Moran's I indicate a positive relationship - spatial clustering of similar values, while negative values indicate spatial dispersion. In this section, we will calculate Moran's I using the [libpysal](#) library to assess the spatial autocorrelation of country area sizes among European countries. We rely on the previously prepared European data set as input data.

We start by importing the necessary libraries, including numpy and We define the spatial weights matrix using the [Queen contiguity](#) method, which considers two regions as neighbors if they share either a boundary or a vertex. This is done using `ps.weights.Queen.from_dataframe` function. A spatial weights matrix defines the spatial structure of a dataset by indicating which observations are neighbors and how strongly they are connected and serves as the primary spatial input for our autocorrelation computations.

Next, we calculate Moran's I for the area column of our GeoDataFrame `gdf_europe` using the Moran class from [esda](#) (ESDA: Exploratory Spatial Data Analysis) module. The Moran function takes the variable of interest (area) from the GeoDataFrame and the spatial weights matrix as inputs and computes Moran's I statistic, which directly measures the spatial autocorrelation in the dataset.

Finally, we print the results, which include Moran's I value and its associated p-value. The p-value helps to determine the statistical significance of the observed spatial autocorrelation.

In

```
1 # Library import
2 import geopandas as gpd
3 import libpysal as ps
4 from esda.moran import Moran
5
6 # Define the spatial weights matrix using the Queen contiguity method
7 w =
8
9 # Calculate Moran's I for the area column in the GeoDataFrame
10 moran = Moran(gdf_europe['area'], w)
11
12 # Print Moran's I and its p-value
13 print(f'Moran's I: p-value:
14
15 # Display the results
16
```

```
('WARNING: ', 28, ' is an island (no neighbors)')
Moran's I: 0.42902696545113683, p-value: 0.005
```

Out

pop_est

361313.0

| | |
|------------|---|
| continent | Europe |
| name | Iceland |
| iso_a3 | ISL |
| gdp_md_est | 24188 |
| area | 20.569244 |
| length | 29.433457 |
| geometry | POLYGON ((-14.739637417041607 65.8087482774403... |

Name: 29, dtype: object

The output indicates that one of the geographic units (index 28) is an "island," meaning it has no neighboring units according to the Queen contiguity method. After displaying that specific record, we see that it is Iceland, so the results sound reasonable. Then, we may observe that the calculated Moran's I value is around 0.4 with a p-value lower than 0.01, which suggests a significant positive spatial autocorrelation in the dataset, implying that countries with similar area sizes tend to be clustered together in Europe.

92. Local Spatial Autocorrelation

Local spatial autocorrelation provides insights into the spatial clustering or dispersion of a particular variable at a local level rather than globally. While the global measure describes the whole data set by one single number, the local metrics assign a clustering score to each polygon within the spatial data set.

Local Moran's I (also known as LISA - Local Indicators of Spatial Association) is a commonly used statistic to identify clusters or outliers within the data. In this section, we will calculate and visualize Local Moran's I for the area sizes of European countries using the libpysal and esda libraries. Again, we will use the map prepared for Europe. Then, we call the `Moran_Local` class from the esda library to calculate Local Moran's I for the area attribute of the `gdf_europe` GeoDataFrame. This function takes the variable of interest and the spatial weights matrix as inputs. Then, we create a new column, in the GeoDataFrame where we store the calculated LISA values. Finally, we plot and display the results showing the spatial distribution of Local Moran's I across Europe.

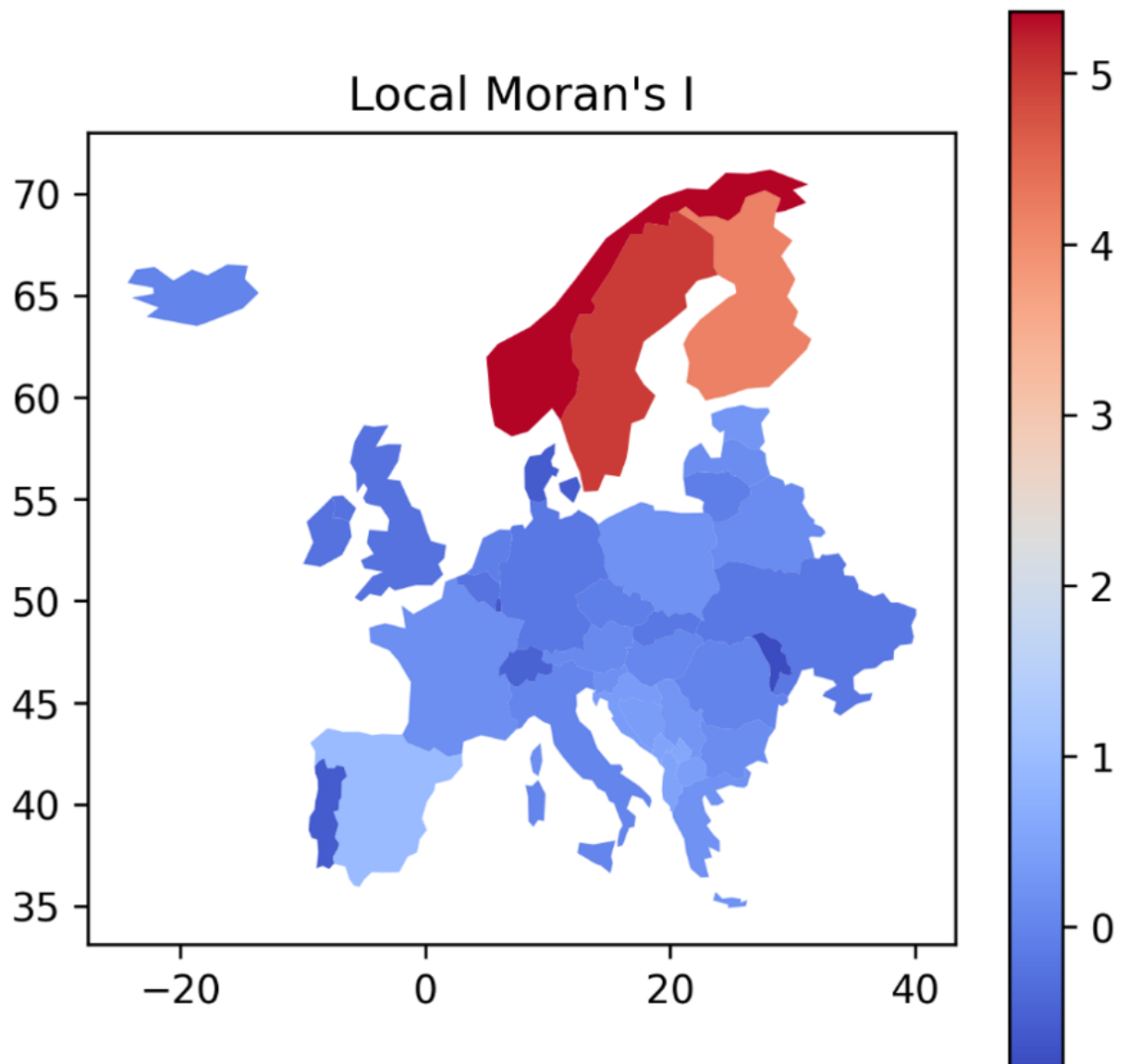
In

```
1 # Library import
2 import geopandas as gpd
3 import matplotlib.pyplot as plt
4 import libpysal as ps
5 from esda.moran import Moran_Local
6
7 # Calculate Local Moran's I
8 w =
```

```
9 moran_local = Moran_Local(gdf_europe['area'], w)

10
11 # Store the LISA values in the GeoDataFrame
12 gdf_europe['moran_local'] =
13
14 # Plot the LISA values
15
16 Moran's I")
17
18
19 # Sort the GeoDataFrame by the LISA values
20 gdf_europe_sorted =
21
22
```

('WARNING: ', 28, ' is an island (no neighbors)')



| | pop_est | continent | name | iso_a3 | gdp_md_est | area | length | geometry | moran_local |
|----|------------|-----------|---------|--------|------------|-----------|------------|---|-------------|
| 1 | 5347896.0 | Europe | Norway | NOR | 403336 | 90.496255 | 127.160983 | POLYGON ((29.39955 69.15692, 28.59193 69.06478... | 5.364920 |
| 3 | 10285453.0 | Europe | Sweden | SWE | 530883 | 79.446214 | 46.690995 | POLYGON ((11.46827 59.43239, 12.30037 60.11793... | 5.001308 |
| 31 | 5520314.0 | Europe | Finland | FIN | 269296 | 63.782393 | 45.337372 | POLYGON ((28.44594 68.36461, 29.97743 67.69830... | 4.157966 |

The map of Local Moran's I indicate significant local spatial autocorrelation in the area sizes of European countries, with Sweden, Finland, and Norway showing the highest positive values, indicating clusters of large areas, which we can even confirm at first glance by observing the significantly larger size of the neighboring Scandinavian countries. In contrast, regions with lower or

negative values suggest areas where the surrounding countries have more varied sizes, highlighting spatial heterogeneity across continental Europe.

93. Spatial Feature Generation

Spatial feature generation refers to the step in geospatial analysis that involves creating new attributes attached to spatial units based on the spatial relationships between the studied geographic entities. These features can provide valuable insights into spatial patterns and dependencies, which are essential for advanced geospatial analyses and machine learning models. In this section, we will demonstrate how to generate a new spatial feature, such as the number of neighboring countries.

We start by importing the necessary libraries and focus on African countries throughout this section. Then, we perform a spatial join to identify intersecting (neighboring) countries. The `gpd.sjoin` function is used to find countries that share borders, where the cases when country is joined with itself are filtered out to avoid self-joins. We then count the number of neighbors for each country using the `groupby` method and store these counts in a DataFrame we call `This count represents the number of neighboring countries for each country`.

Additionally, we note that from the previous sections, we already have a few additional spatial features prepared, namely, the area and length of the polygons and the LISA clustering values for every African country.

In

```
1 # Library import
2 import geopandas as gpd
3 import libpysal as ps
4 from esda.moran import Moran_Local
5 import pandas as pd
```

```
7 # Spatial join to find intersecting (neighboring) countries
8 gdf_neighbors =
9
10
11
12 # Exclude self-joins (where a country is joined with itself)
13 gdf_neighbors =
14             != gdf_neighbors['index_right']]
15
16 # Count the number of neighbors for each country
17 neighbor_counts =
18 df_neighbor_counts =
19
20
21 # Add the neighbor count as a new column in the original GeoDataFrame
22 if 'neighbor_count' not in gdf_africa:
23     gdf_africa =
24
25
26
27 # Replace NaN values with 0 (for countries with no neighbors)
28 gdf_africa['neighbor_count'] =
29
30 # Calculate area and length of each geometry

31 gdf_africa['area'] =
32 gdf_africa['length'] =
33
34 # Compute the Moran I LISA values in the GeoDataFrame
35 w =
36 moran_local = Moran_Local(gdf_africa['area'], w)
```

```

37 gdf_africa['moran_local'] =
38
39 # Display the first 3 rows
40
41
42 # Print the spatial features
43 keys =
44 print("The Africa GeoDataFrame contains the followig features:")
45 for feature in keys:
46     if feature not In 'name', 'iso_a3', 'geometry':
47         print(feature)

```

| | pop_est | continent | name | iso_a3 | gdp_md_est | geometry | area | length | neighbor_count | moran_local |
|----|------------|-----------|-----------------|--------|------------|--|------------|-----------|----------------|-------------|
| 1 | 58005463.0 | Africa | Tanzania | TZA | 63177 | POLYGON ((33.90371 0.95000, 34.07262 1.05982... | 76.301964 | 37.260671 | 8 | 0.001039 |
| 2 | 603253.0 | Africa | W. Sahara | ESH | 907 | POLYGON ((8.66559 27.65643, 8.66512 27.58948... | 8.603984 | 27.662143 | 3 | 1.089056 |
| 11 | 86790567.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((20.34000 4.49998, 20.51999 5.41998... | 189.515232 | 75.277618 | 9 | 0.323053 |

The Africa GeoDataFrame contains the followig features:

pop_est

gdp_md_est

area

length

neighbor_count

moran_local

The resulting spreadsheet shows the values of all spatial and non-spatial features generated above.

94. OLS Regression on Spatial Data

Regression analysis, including Ordinary Least Squares (OLS), is a fundamental statistical toolset for understanding and quantifying the relationships between variables. It allows us to make predictions and infer causal relationships by examining how changes in the independent variables affect the dependent variable. In the context of spatial data, OLS regression can help identify spatial patterns and dependencies, making it a valuable tool for geospatial analysis. In this section, we will demonstrate how to perform OLS regression on spatial data using the `geopandas` and [statsmodels](#) libraries. In particular, we will use several previously derived country-level features of Africa to estimate their GDP levels stored in the `gdp_md_est` data column. We start by importing the necessary libraries: `geopandas` for handling spatial data, `statsmodels` for performing OLS regression, and `matplotlib.pyplot` for visualization. Next, we prepare the data for regression by selecting the dependent and independent variables. In this example, we use the `gdp_md_est` (population estimate) as the dependent variable and `pop_est` as the independent variables (features). We fit an OLS regression model using the `OLS` class from `statsmodels`. After fitting the model, we print the summary of the regression results to interpret the coefficients and their statistical significance. Finally, we visualize the regression results by plotting the observed vs. predicted GDP values.

In

```
1 # Library import
2 import geopandas as gpd

3 import statsmodels.api as sm
4 import matplotlib.pyplot as plt
```

```

5 import pandas as pd
6
7 # Displaying the input data to confirm we have all the features
8

```

| | pop_est | continent | name | iso_a3 | gdp_md_est | geometry | area | length | neighbor_count | moran_local |
|----|------------|-----------|-----------------|--------|------------|--|-----------|-----------|----------------|-------------|
| 1 | 58005463.0 | Africa | Tanzania | TZA | 63177 | POLYGON ((33.90371 -0.95000, 34.07262 -1.05902... | 76301964 | 37.260671 | 8 | 0.001039 |
| 2 | 603253.0 | Africa | W. Sahara | ESH | 907 | POLYGON ((-0.66559 27.65643, -0.66512 27.50948... | 8603904 | 27.662143 | 3 | -1.089056 |
| 11 | 86790567.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((29.34000 -1.149990, 29.51999 -5.11990... | 109515232 | 75.277618 | 9 | -0.323053 |

In

```

1 #Prepare data for regression
2 #Dependent variable
3 y = gdf_africa['gdp_md_est']
4
5 #Independent variable
6 X = gdf_africa[['neighbor_count', 'area', 'moran_local', 'pop_est' ]]
7
8 #Add a constant term for the intercept
9 X =
10
11 #Fit the OLS regression model
12 model =
13
14 #Print the regression results
15

```

```

=====
                        OLS Regression Results
=====
Dep. Variable:          gdp_md_est      R-squared:                0.740
Model:                  OLS             Adj. R-squared:           0.717
Method:                 Least Squares    F-statistic:              32.07
Date:                   Tue, 02 Jul 2024  Prob (F-statistic):      1.18e-12
Time:                   15:02:05         Log-Likelihood:           -606.93
No. Observations:       50              AIC:                     1224.
Df Residuals:           45              BIC:                     1233.
Df Model:                4
Covariance Type:        nonrobust
=====

```

| | coef | std err | t | P> t | [0.025 | 0.975] |
|----------------|------------|----------|--------|-------|-----------|-----------|
| const | 3.8e+04 | 1.7e+04 | 2.235 | 0.030 | 3759.587 | 7.22e+04 |
| neighbor_count | -1.392e+04 | 4527.248 | -3.074 | 0.004 | -2.3e+04 | -4798.333 |
| area | 402.0043 | 222.919 | 1.803 | 0.078 | -46.977 | 850.986 |
| moran_local | 5779.1621 | 1.22e+04 | 0.474 | 0.638 | -1.88e+04 | 3.04e+04 |
| pop_est | 0.0020 | 0.000 | 9.208 | 0.000 | 0.002 | 0.002 |

```

=====
Omnibus:                50.621      Durbin-Watson:            2.171
Prob(Omnibus):           0.000      Jarque-Bera (JB):         418.152
Skew:                    2.295      Prob(JB):                 1.58e-91
Kurtosis:                16.403     Cond. No.:                1.13e+08
=====

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

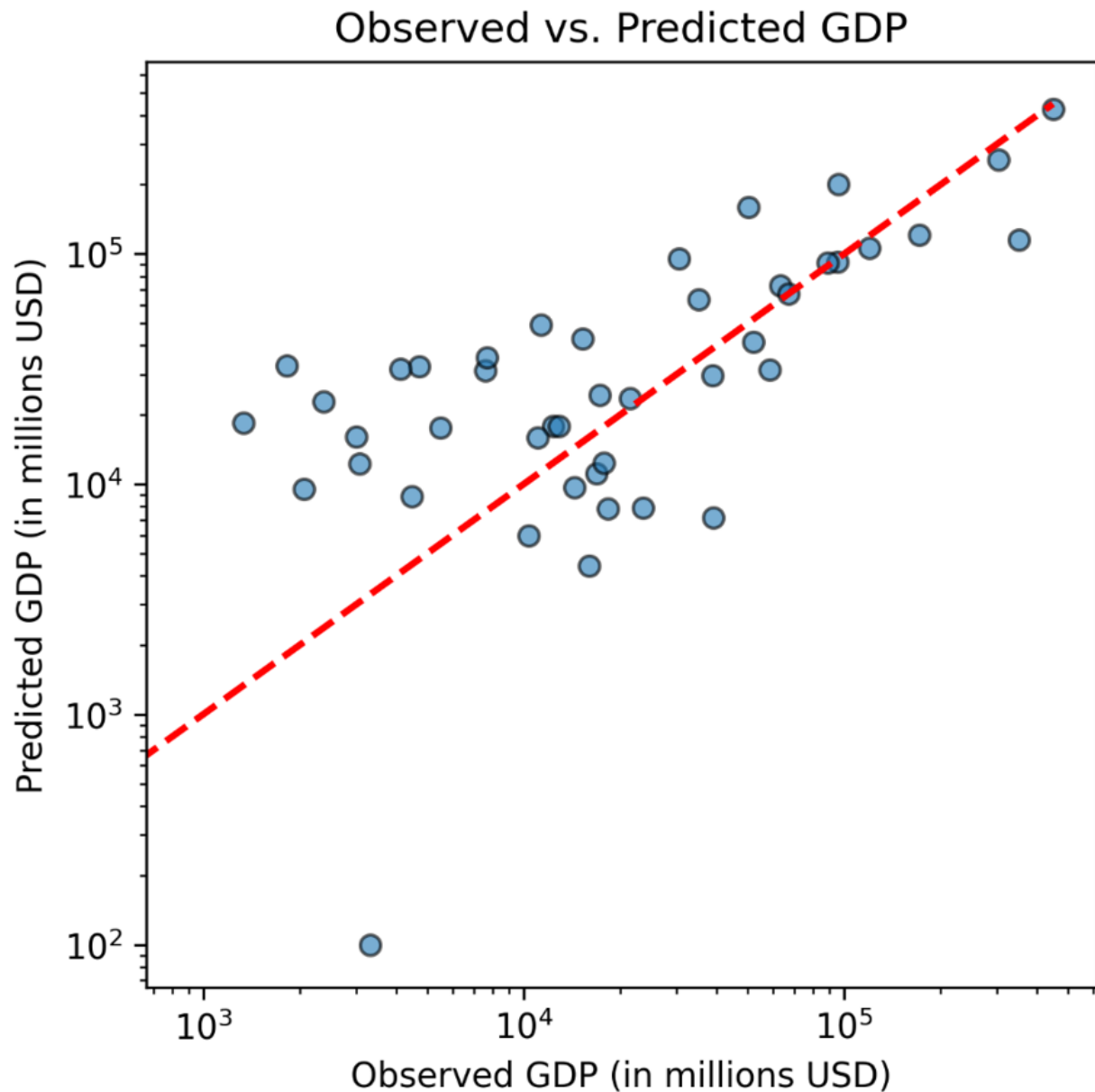
[2] The condition number is large, 1.13e+08. This might indicate that there are

strong multicollinearity or other numerical problems.

In

- 1 # Plot observed vs. predicted values
- 2 # Predict GDP using the regression
- 3 #model and add predictions to the GeoDataFrame
- 4 gdf_africa['predicted_gdp'] =

```
5
6 # Plot observed vs. predicted values
7 f, ax =
8 gdf_africa['predicted_gdp'],
9
10 GDP (in millions USD)')
11 GDP (in millions USD)')
12 vs. Predicted GDP')
13
14 # Plot the line y = x for reference
15 min_val =
16
17 max_val =
18
19 max_val], [min_val, max_val],
20
21
22 # Setting up a log scale on both axis
23
24
25
26 # For future analysis, we wont need this column
27 gdf_africa = gdf_africa[['predicted_gdp']]
28
29 # Show the plot
30
```



The regression results show that the model has an R-squared value of 0.740, meaning it explains 74.0% of the variance in GDP estimates. Among the predictors, the population estimate is highly significant ($p < 0.000$), indicating a strong positive relationship with GDP. On the other hand, the `neighbor_count` has a significant negative coefficient, suggesting that having more neighbors is associated with lower GDP estimates. The model is not statistically significant in this model.

It is important to note that the condition number is large ($1.13e+08$), which indicates potential issues with multicollinearity or other numerical problems in the model. While we focus on the core technique here, it is recommended to further explore and address multicollinearity to ensure the reliability of the regression results when working on a real-life project.

95. Spatial Regression Models

Spatial regression models are advanced techniques used to directly account for spatial dependencies in data, improving the accuracy and reliability of predictions. This section will cover two common spatial regression models: the Spatial Lag Model (SLM) and the Spatial Error Model (SEM). These models incorporate spatial weights matrices to capture spatial relationships between observations:

Spatial Lag Model Accounts for the influence of neighboring values on the dependent variable.

Spatial Error Model Accounts for spatial autocorrelation in the error terms of the regression.

Here, again, we will work with the country-level Africa map, which has several features already generated. First, we create a spatial weights matrix using the Queen contiguity method, which we introduced in an earlier section and defines neighbors as observations sharing a common edge or vertex. This matrix is then transformed into a row-standardized form. Next, we define the dependent and independent variables for our regression models. The target variable is the dependent variable, while features represent the independent variables. In this example, we will take the country's GDP level as the target variable, which we will try to predict using the previously generated spatial features.

We fit the Spatial Lag Model (SLM) using the `ML_Lag` class from the `spreg` module, which incorporates spatial dependencies by including a spatially lagged dependent variable in the model. The results are summarized and printed.

Finally, we fit the Spatial Error Model (SEM) using the ML_Error class from the spreg module. This model accounts for spatial dependencies in the error terms, providing a robust framework for regression analysis in the presence of spatial autocorrelation. The results of this model are also summarized and printed.

In

```
1 # Library imports
2 import geopandas as gpd
3 import libpysal as ps
4
5 # Displaying the input data to confirm we have all the features
6
```

| | pop_est | continent | name | iso_a3 | gdp_md_est | geometry | area | length | neighbor_count | moran_local |
|----|------------|-----------|-----------------|--------|------------|---|-----------|----------|----------------|-------------|
| 1 | 58005463.0 | Africa | Tanzania | TZA | 63177 | POLYGON ((33.90371 -0.95000, 34.07262 -1.05902... | 76301964 | 37260671 | 0 | 0.001039 |
| 2 | 603253.0 | Africa | W. Sahara | ESH | 907 | POLYGON ((-8.66559 27.65643, -8.66512 27.58940... | 8603904 | 27662143 | 3 | -1.089056 |
| 11 | 86790567.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((29.34000 -4.49998, 29.51999 -5.41998... | 189515232 | 75277618 | 9 | -0.323053 |

In

```
1 # Compute the Moran I LISA values in the GeoDataFrame
2 w =
3 = 'r'
```

In

```
1 # Spatial Lag Model (SLM)
```



```

2 from spreg import ML_Lag
3
4 # Define dependent and independent variables
5 target = 'gdp_md_est'
6 features = ['neighbor_count', 'area', 'moran_local', 'pop_est' ]
7 y =
8
9
10 # Fit the Spatial Lag Model
11 lag_model = ML_Lag(y, X, w,
12

```

REGRESSION RESULTS

SUMMARY OF OUTPUT: MAXIMUM LIKELIHOOD SPATIAL LAG (METHOD = FULL)

| | | | | |
|--------------------------|---|-----------------|-------------------------|----------|
| Data set | : | unknown | | |
| Weights matrix | : | unknown | | |
| Dependent Variable | : | gdp_md_est | Number of Observations: | 50 |
| Mean dependent var | : | 48828.0000 | Number of Variables | 6 |
| S.D. dependent var | : | 89671.0462 | Degrees of Freedom | 44 |
| Pseudo R-squared | : | 0.7431 | | |
| Spatial Pseudo R-squared | : | 0.7396 | | |
| Log likelihood | : | -606.7002 | | |
| Sigma-square ML | : | 2024418543.6593 | Akaike info criterion : | 1225.400 |
| S.E of regression | : | 44993.5389 | Schwarz criterion | 1236.872 |

| Variable | Coefficient | Std.Error | z-Statistic | Probability |
|----------------|--------------|-------------|-------------|-------------|
| CONSTANT | 46291.44090 | 17918.64238 | 2.58342 | 0.00978 |
| neighbor_count | -14953.63965 | 4273.32600 | -3.49930 | 0.00047 |
| area | 456.74445 | 212.08670 | 2.15357 | 0.03127 |
| moran_local | 2070.94658 | 11819.89823 | 0.17521 | 0.86092 |
| pop_est | 0.00195 | 0.00021 | 9.36586 | 0.00000 |
| W_gdp_md_est | -0.08489 | 0.14788 | -0.57403 | 0.56595 |

===== END OF REPORT =====

In

```

1 # Spatial Error Model (SEM)
2 from spreg import ML_Error
3
4 # Fit the Spatial Error Model
5 error_model = ML_Error(y, X, w,
6

```

REGRESSION RESULTS

SUMMARY OF OUTPUT: ML SPATIAL ERROR (METHOD = full)

```

-----
Data set          :      unknown
Weights matrix    :      unknown
Dependent Variable :  gdp_md_est
Mean dependent var :  48828.0000
S.D. dependent var :  89671.0462
Pseudo R-squared   :    0.7389
Log likelihood     :   -606.6713
Sigma-square ML    : 2019314062.5344
S.E of regression  :  44936.7785

Number of Observations:      50
Number of Variables   :       5
Degrees of Freedom    :      45

Akaike info criterion :  1223.343
Schwarz criterion     :  1232.903

```

| Variable | Coefficient | Std.Error | z-Statistic | Probability |
|----------------|--------------|-------------|-------------|-------------|
| CONSTANT | 45037.02522 | 16068.84183 | 2.80275 | 0.00507 |
| neighbor_count | -15847.20521 | 4216.06887 | -3.75876 | 0.00017 |
| area | 481.83184 | 198.57869 | 2.42640 | 0.01525 |
| moran_local | 2190.28213 | 11107.53719 | 0.19719 | 0.84368 |
| pop_est | 0.00194 | 0.00020 | 9.57190 | 0.00000 |
| lambda | -0.11619 | 0.20108 | -0.57782 | 0.56338 |

===== END OF REPORT =====

The Spatial Lag Model results indicate a good fit with a pseudo R-squared value of 0.7431, suggesting that the model explains a significant portion of the variance in GDP estimates. The coefficient for the spatially lagged dependent variable is not statistically significant, indicating that spatial dependence might not be strong in this model. The pop_est variable is highly significant, showing a strong positive relationship with GDP. The neighbor_count and area variables are also significant, with neighbor_count

having a negative relationship and area a positive one. However, the moran_local variable is not significant.

The Spatial Error Model (SEM) also demonstrates a good fit with a pseudo R-squared of 0.7389. The pop_est variable remains highly significant, while neighbor_count and area are significant as well, with neighbor_count showing a negative relationship and area a positive one. The moran_local variable is not significant in this model either. “lambda’ in code formatting, please, which indicates the spatial error term, is not significant, suggesting that spatial error dependence might not be strong.”

Overall, both models highlight the significant impact of population estimates on GDP, but the significance and impact of other predictors such as neighbor_count and area vary between the models. The SLM shows the importance of considering spatial relationships directly through the lagged dependent variable, whereas the SEM accounts for spatial error dependence, which appears to be less significant in this case.

Further exploration of multicollinearity is recommended to ensure the reliability of these regression results. By addressing multicollinearity, ensuring proper model specification, and including further spatial features the reliability, predictive power, and interpretability of these spatial models can be improved.

96. Spatial Random Forest

Random Forest is a popular machine learning algorithm that operates by constructing multiple decision trees during training and outputting the mean prediction of the individual trees. When adapting Random Forest for spatial data, it is essential to account for spatial dependencies and autocorrelations. This can be done by incorporating spatial attributes and features derived from the spatial relationships between data points. In this section, we will demonstrate how to train and evaluate a Spatial Random Forest model using the world map's Africa subset, focusing on predicting GDP level.

In the code cell below, we first take care of the necessary imports and data preparation steps. To conduct the regression, we use the `RandomForestRegressor` from the [sklearn](#) library with 100 estimators. Then, we split the dataset into training and testing sets to evaluate the model's performance using mean-squared error and R-squared metrics. Finally, we visualize the observed versus predicted population densities and plot the importance of each feature to understand the contribution of each feature to the model's predictions.

In

```
1 # Importing libraries
2 import geopandas as gpd
3 import numpy as np
4 import pandas as pd
5 from sklearn.ensemble import RandomForestRegressor
6 from sklearn.model_selection import train_test_split
7 from sklearn.metrics import mean_squared_error, r2_score
```

```
8 import matplotlib.pyplot as plt
```

```
9
```

```
10 # Displaying the input data to confirm we have all the features
```

```
11
```

| | pop_est | continent | name | iso_a3 | gdp_md_est | geometry | area | length | neighbor_count | moran_local |
|----|------------|-----------|-----------------|--------|------------|---|-----------|----------|----------------|-------------|
| 1 | 58005463.0 | Africa | Zanzania | TZA | 63177 | POLYGON ((33.80371 -0.95000, 34.07262 -1.05982... | 76305964 | 37260671 | 8 | 0.001039 |
| 2 | 603293.0 | Africa | W. Sahara | ESH | 907 | POLYGON ((8.66559 27.65643, 8.66512 27.58948... | 8603984 | 27662143 | 3 | 1.069056 |
| 11 | 86790567.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((29.34000 4.49998, 29.51999 5.41998... | 189515232 | 75277618 | 9 | 0.323053 |

```
In
```

```
1 # Compute the Moran I LISA values in the GeoDataFrame
```

```
2 w =
```

```
3 = 'r'
```

```
4
```

```
5 # Prepare the feature matrix and target vector
```

```
6 features = gdf_africa[['neighbor_count', 'area', 'moran_local', 'pop_est']]
```

```
7 target = gdf_africa['gdp_md_est']
```

```
8
```

```
9 # Handle missing values
```

```
10 features =
```

```
11 target =
```

```
12
```

```
13 # Split data into training and testing sets
```

```
14 X_train, X_test, y_train, y_test = train_test_split(features,
```

```
15                                     target,
```

```
16
```

```
17
```

```
18
```

```
19 # Train the Random Forest model
20 rf =
21 y_train)
22
23 # Make predictions
24 y_pred =
25
26 # Evaluate the model
27 mse = mean_squared_error(y_test, y_pred)
28 r2 = r2_score(y_test, y_pred)
29 print(f'Mean Squared Error: {mse}')
30 print(f'R-squared: {r2}')
```

Mean Squared Error: 4281842457.6505995

R-squared: 0.671629891746841

In

```
1 # Plot observed vs. predicted values
2 f, ax = plt.subplots(1, 6))
3 y_pred,
4 GDP (in millions USD))
5 GDP (in millions USD))
6 vs. Predicted GDP')
7
8 # Plot the line y = x for reference
9 min_val =
10 max_val =
11 max_val], [min_val, max_val],
```

12

13

14 # Setting up a log scale on both axes

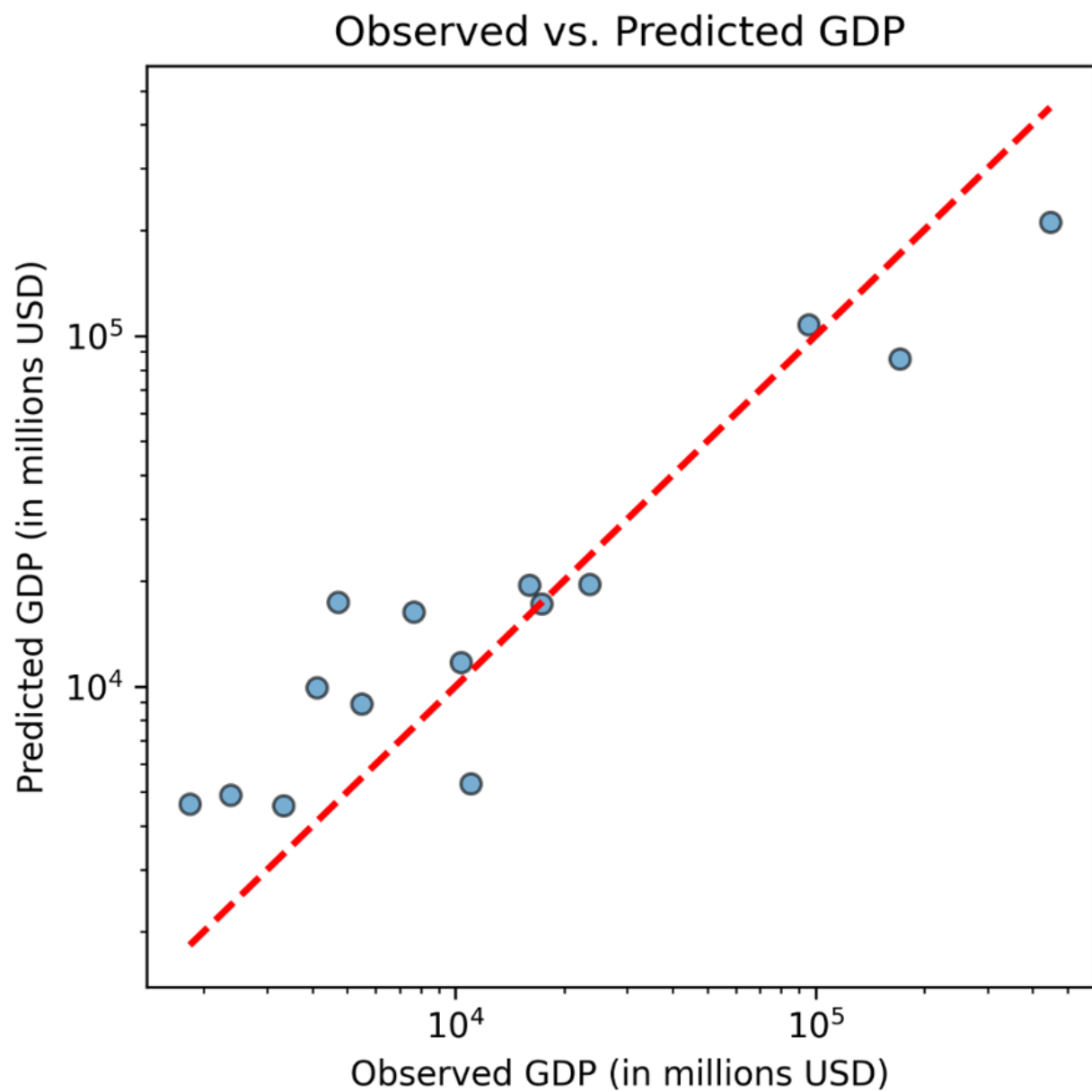
15

16

17

18 # Show the plot

19



In

1 # Calculate feature importances

2 importances =

3 feature_names =

4 indices =

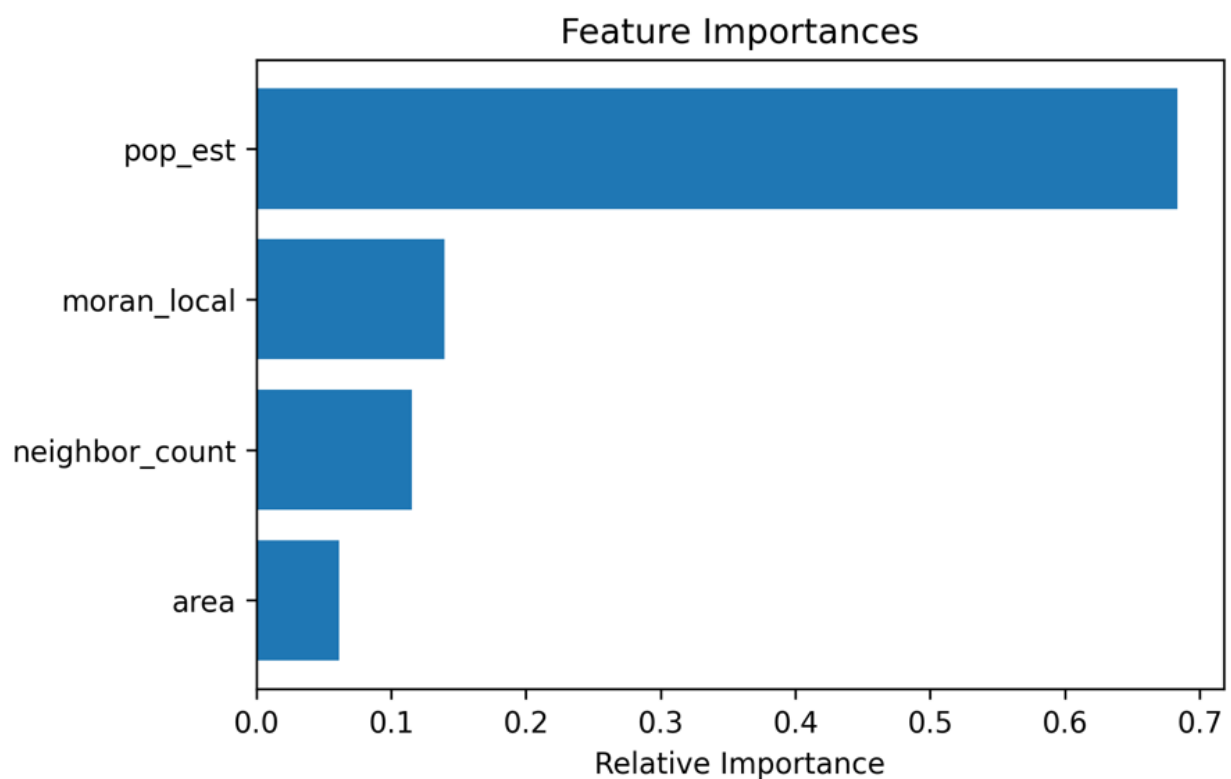
5

6 # Plot feature importances


```

7 f, ax = 1, 4))
8 importances[indices],
9
10 for i in indices])
11 Importance')
12 Importances')
13
14 # Show the plot
15

```



On the one hand, the final model performance on the scatter plot shows a reasonably good fit between the predicted and original values, with an R-squared value of approximately 0.67. This indicates that about 67% of the variance in population density is explained by the model. The Mean Squared Error (MSE) is also provided, indicating the average squared difference

between observed and predicted values. While there are some outliers, the majority of the data points cluster near the diagonal line, suggesting the model's predictions are generally accurate.

On the other hand, we can zoom in and compare the predictive power of the different features, as shown in the second graph. This plot reveals that the estimated population is the most influential predictor of GDP followed by the local Moran's I statistic and the neighbor count. This ranking of feature importance helps in understanding which variables contribute most to predicting population density, providing valuable insights for further analysis and decision-making.

97. Comparing Spatial Regression Models

In this section, we will compare the goodness-of-fit metrics for four regression models: OLS (Ordinary Least Squares), Spatial Lag Model (SLM), Spatial Error Model (SEM), and Random Forest Regression. These metrics include R-squared, [AIC \(Akaike Information\)](#) and Log-Likelihood, which help evaluate the models' performance and suitability for spatial data analysis.

First, we import the necessary libraries. Additionally, we will rely on the previously prepared features of the African map GeoDataFrame. Then, we build a spatial weights matrix using Queen contiguity with `ps.weights.Queen.from_dataframe` and transform it to row-standardized weights with

Next, we define the target variable and the features for the regression models. We ensure that our data is prepared correctly by dropping any rows with missing values in the target and feature columns using `dropna` command. We then extract the dependent variable stored in `y` and the independent variables stored in `X` from the input GeoDataFrame.

For the OLS model, we add a constant to the feature matrix using `sm.add_constant` and fit the model using `sm.OLS.fit` function. For the Spatial Lag Model, we fit the model using `sm.SLM` and for the Spatial Error Model, we use `sm.SEM`. Finally, we fit a Random Forest model using `RandomForestRegressor` from

We then extract the R-squared, AIC, and Log-Likelihood values from each model. The OLS model's metrics are obtained from the `ols_model` object, while the Spatial Lag and Error models' metrics are extracted from the respective `lag_model` and `error_model` objects. For the Random Forest model, we use the `mean_squared_error` function from `sklearn.metrics` to compute the R-squared value.

Finally, we print a table comparing the goodness-of-fit metrics for the OLS, SLM, SEM, and Random Forest models.

In

```
1 # Import necessary libraries
2 import geopandas as gpd
3 import libpysal as ps
4 import statsmodels.api as sm
5 from spreg import ML_Lag, ML_Error
6 from sklearn.ensemble import RandomForestRegressor
7 from sklearn.metrics import mean_squared_error, r2_score
8
9 # Displaying the input data to confirm we have all the features
10
11
12 # Compute the Moran I LISA values in the GeoDataFrame
13 w =
14 moran_local = Moran_Local(gdf_africa['area'], w)
15 gdf_africa['moran_local'] =
16 = 'r'
17
18 # Define target variable and features
19 target = 'gdp_md_est'
20 features = ['neighbor_count', 'area', 'moran_local', 'pop_est']
21
22 # Prepare the data
23
24
25 gdf_africa = + features)
26 y =
27 X =
```

| | pop_est | continent | name | iso_a3 | gdp_md_est | geometry | area | length | neighbor_count | moran_local |
|----|------------|-----------|-----------------|--------|------------|---|-----------|----------|----------------|-------------|
| 1 | 58005463.0 | Africa | Egypt | EGY | 63177 | POLYGON ((33.90371 -0.95000, 34.07262 -1.05982... | 76301964 | 37260671 | 8 | 0.001039 |
| 2 | 603253.0 | Africa | W. Sahara | ESH | 907 | POLYGON ((8.66559 27.65643, 8.66512 27.58948... | 8603984 | 27662145 | 5 | 1.089056 |
| 11 | 86790367.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((29.34000 4.49998, 29.51999 5.41998... | 189515252 | 75277618 | 9 | 0.423053 |

In

```

1 # Add a constant for OLS
2 X_ols =
3
4 # Fit the OLS model
5 ols_model =
6
7 # Fit the Spatial Lag Model
8 lag_model = ML_Lag(y, X, w,
9
10 # Fit the Spatial Error Model
11 error_model = ML_Error(y, X, w,
12
13 # Fit the Random Forest model
14 X_train, X_test, y_train, y_test = train_test_split(X, y,
15
16
17 rf_model =
18
19 y_train)
20 y_pred_rf =
21 rf_r2 = r2_score(y_test, y_pred_rf)
22 rf_mse = mean_squared_error(y_test, y_pred_rf)
23
24 # Extract OLS goodness of fit metrics
25 ols_r2 =

```

```
25 ols_aic =
26 ols_loglik =
27
28 # Extract Spatial Lag Model goodness of fit metrics
29 lag_r2 =
30 lag_aic =
31 lag_loglik =
32
33 # Extract Spatial Error Model goodness of fit metrics
34 error_r2 =
35 error_aic =
36 error_loglik =
37
38 # Print comparison of goodness of fit
39 print("\nGoodness of Fit Comparison:")
40 print(f'{"Model":<20} {"R-squared":<15} {"AIC":<15} \
41       {"Log-Likelihood":<15}')"
42
43 print(f'{"OLS":<20} {ols_r2:<15.4f} {ols_aic:<15.4f} \
44       {ols_loglik:<15.4f}')"
45
46 print(f'{"Spatial Lag Model":<20} {lag_r2:<15.4f} \
47       {lag_aic:<15.4f} {lag_loglik:<15.4f}')"
48
49 print(f'{"Spatial Error Model":<20} {error_r2:<15.4f} \
50       {error_aic:<15.4f} {error_loglik:<15.4f}')"
51
52 print(f'{"Random Forest":<20} {rf_r2:<15.4f} \
53       {"N/A":<15} {"N/A":<15}')
```

Goodness of Fit Comparison:

| Model | R-squared | AIC | Log-Likelihood |
|---------------------|-----------|-----------|----------------|
| OLS | 0.7403 | 1223.8576 | -606.9288 |
| Spatial Lag Model | 0.7431 | 1225.4004 | -606.7002 |
| Spatial Error Model | 0.7389 | 1223.3425 | -606.6713 |
| Random Forest | 0.6716 | N/A | N/A |

The goodness-of-fit comparison for the OLS, Spatial Lag Model (SLM), Spatial Error Model (SEM), and Random Forest model reveals key insights into their performance with spatial data. The R-squared values for the OLS, SLM, SEM, and Random Forest models are 0.7403, 0.7431, 0.7389, and 0.6713, respectively. While the OLS, SLM, and SEM models explain about 74% of the variance in the dependent variable, GDP, the Random Forest model explains only about 67%, suggesting it still captures a substantial portion but slightly less than the others.

Examining the Akaike Information Criterion (AIC), the SEM has the lowest value (1223.3425), suggesting it offers the best balance between fit and complexity. The Log-Likelihood values, higher for the SEM and SLM, also indicate a better fit compared to the OLS model. The Random Forest model, being a non-parametric method, does not provide AIC and Log-Likelihood values but demonstrates its usefulness through feature importance and predictive accuracy.

In summary, the SEM emerges as the best model based on AIC and Log-Likelihood, making it suitable when accounting for spatial error autocorrelation is crucial. The SLM is slightly better than OLS, useful when spatial lags in the dependent variable are significant. The OLS model remains a robust baseline for simpler analyses without strong spatial dependencies. The Random Forest model, while slightly less precise in this context, offers valuable insights into feature importance and is beneficial for capturing non-linear relationships and interactions.

Model Selection Guidance:

Use when spatial error autocorrelation is expected, providing the best fit and balance.

Opt for this when spatial lags in the dependent variable are important.

Ideal for simpler models without significant spatial effects, offering ease of interpretation and implementation.

Random Choose this to understand the importance of features and capture non-linear relationships, especially when interpretability and interactions between variables are crucial.

98. Hotspot Analysis

Hotspot analysis in spatial analytics is a statistical technique used to identify areas with a high concentration of events or values, as well as areas with low concentrations, often referred to as cold spots. This analysis helps in understanding spatial patterns and is widely used in fields such as crime analysis, epidemiology, environmental science, and urban planning.

One of the most common methods for hotspot analysis is the Getis-Ord Gi statistic, which measures the intensity of clustering for high or low values in a given dataset. The Getis-Ord Gi statistic identifies clusters of high or low values in spatial data, with high scores indicating hotspots and low scores indicating cold spots.

In the example below, we will use the previously introduced esda library and its `G_Local` function to identify population hotspots in Africa using the data prepared earlier. To achieve this, we will use the population level feature as an input for the Getis-Ord Gi* statistic and use Matplotlib to visualize the results.

In

```
1 ### Necessariy imports
2 import geopandas as gpd
3 import numpy as np
4 from esda.getisord import G_Local
5 import matplotlib.pyplot as plt
6 import libpysal as ps
7
```

```

8 # Create spatial weights matrix (e.g., Queen contiguity) for Africa
9 w =

10 = 'r'
11
12 # Example variable (e.g., population estimate)
13 variable =
14
15 # Calculate Getis-Ord Gi* statistic
16 g_local = G_Local(variable, w)
17
18 # Add results to GeoDataFrame
19 gdf_africa['Gi*'] =
20

```

Out

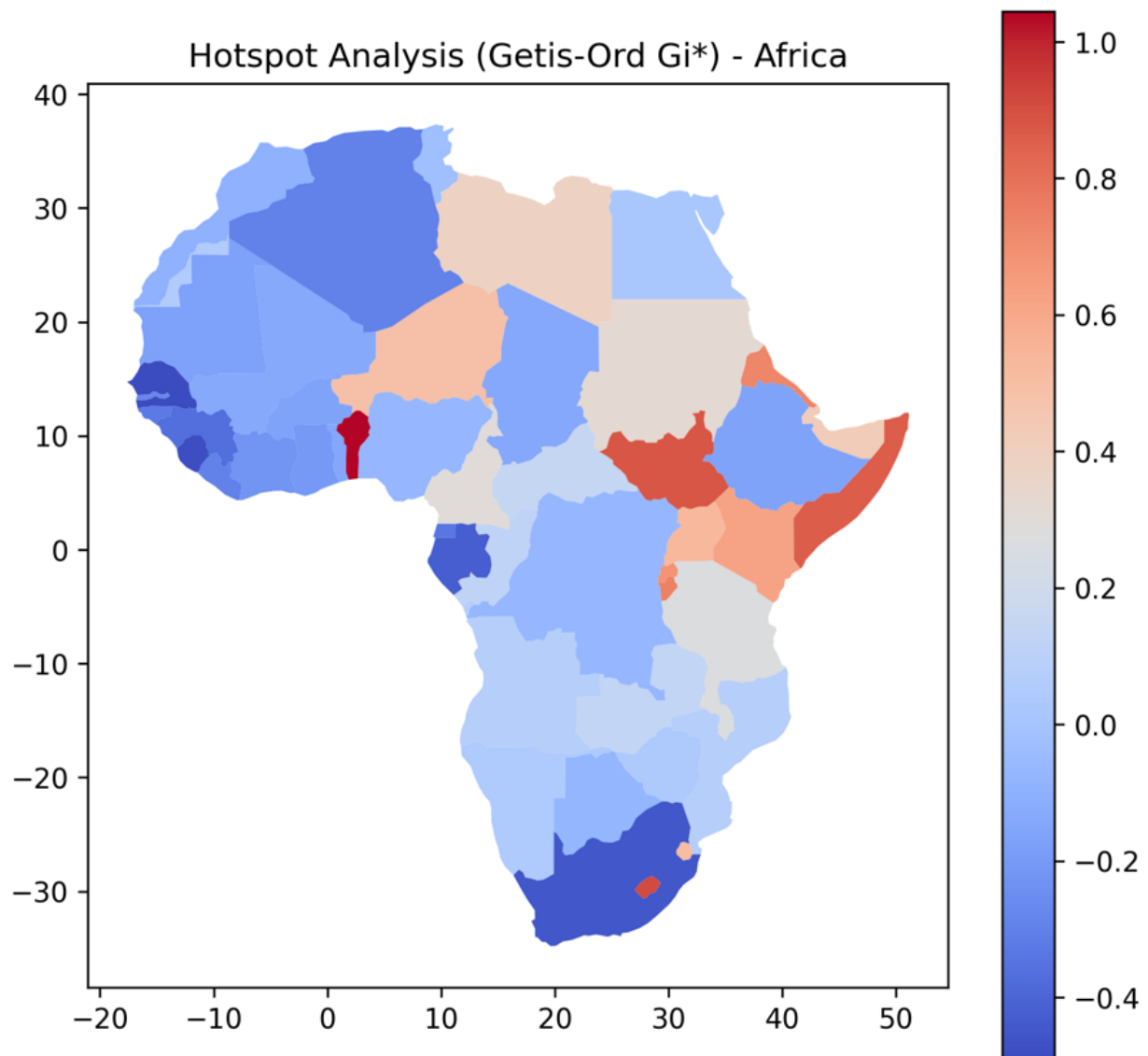
| | pop est | continent | name | iso a3 | gdp md est | geometry | area | length | neighbor count | moran local | Gi* |
|----|------------|-----------|-----------------|--------|------------|---|-----------|-----------|----------------|-------------|----------|
| 1 | 56005463.0 | Africa | Tanzania | TZA | 63177 | POLYGON ((13.90171 -0.95000, 34.07262 -1.05982... | 76301664 | 37.260671 | 8 | 0.001039 | 0.265370 |
| 2 | 603253.0 | Africa | W. Sahara | LSH | 907 | POLYGON ((-8.66659 27.65643, -8.66512 27.58948... | 8603981 | 27.662143 | 3 | -1.089056 | 0.053856 |
| 11 | 86790367.0 | Africa | Dem. Rep. Congo | COD | 50400 | POLYGON ((29.54000 -4.49968, 29.51999 -5.41998... | 189515232 | 75.277618 | 9 | 0.523013 | 0.069750 |

In

```

1 # Plot hotspots for Africa
2 fig, ax = plt.subplots(1, 7))
3
4 Analysis (Getis-Ord Gi*) - Africa")
5

```



The hotspot analysis using the Getis-Ord G_i^* statistic for the population estimates of African countries reveals regions with significantly high or low values compared to their neighbors. On the map above, these are visualized using a red-to-blue color map, which we may interpret as follows.

Red These are identified as hotspots, meaning they have high population estimates and are surrounded by countries with similarly high population estimates. For instance, the regions in East Africa show up mostly in red, indicating a significant clustering of high population estimates.

Blue These are identified as cold spots, meaning they have low population estimates and are surrounded by countries with similarly low population estimates. Several countries in Western and Central Africa appear in blue, indicating significant clustering of low population estimates.

Neutral Regions that do not show strong clustering of high or low values are displayed in neutral colors (light shades). These areas do not have statistically significant spatial patterns of population estimates compared to their neighbors.

This analysis helps in identifying regions with high and low population densities, which can be useful for urban planning, resource allocation, and policy-making.

99. Kernel Density Estimation

Kernel Density Estimation (KDE) is a non-parametric method used to estimate the probability density function of a random variable. In spatial analysis, KDE is often used to identify areas of high concentration (hotspots) or density of events across a geographic area. KDE is useful for visualizing the intensity of point patterns, such as crime incidents, disease outbreaks, or other spatial phenomena.

KDE works by placing a smooth kernel (typically a Gaussian) over each data point and summing the contributions of all kernels to estimate the density at each location in the study area. The result is a continuous surface representing the intensity or density of the points.

In this code, we first extract the coordinates of the centroids to use as anchor points representing each country, which are further enriched by the population estimates from the GeoDataFrame. The `gaussian_kde` function from `scipy.stats` is used to perform the KDE. We then create a grid over the study area to evaluate the density function and plot the resulting density surface. The boundaries of the African countries are plotted on top of the density surface to provide geographic context.

In

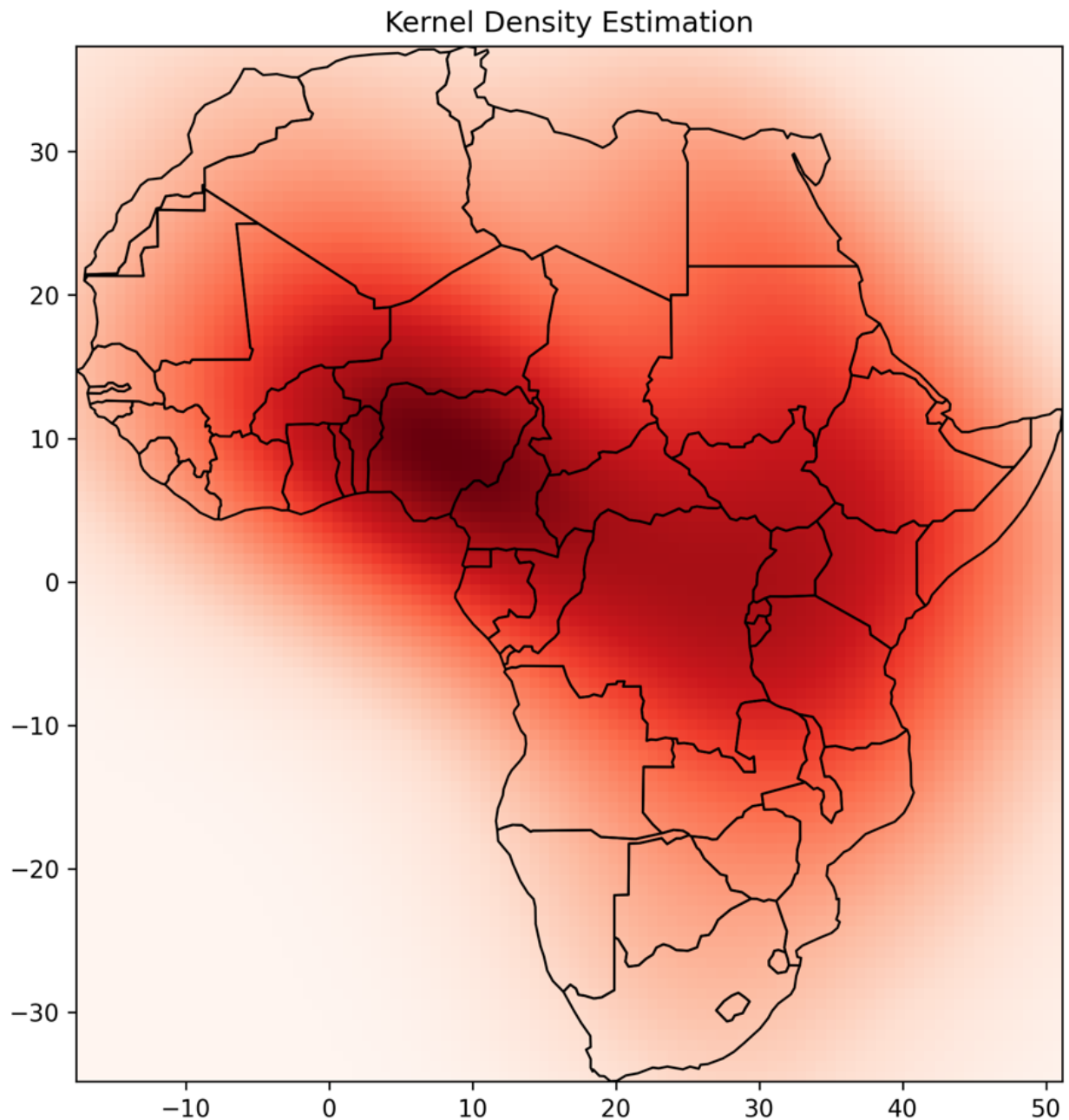
```
1 # Library imports
2 import geopandas as gpd
3 import numpy as np
4 from scipy.stats import gaussian_kde
5 import matplotlib.pyplot as plt
```

```

7 # Extract coordinates of centroids and values for KDE
8 coordinates =
9
10 values = gdf_africa['pop_est']
11
12 # Perform Kernel Density Estimation
13 kde = gaussian_kde(coordinates,
14
15 # Create grid for evaluation
16 minx, miny, maxx, maxy =
17 x, y = miny:maxy:100j]
18 positions =
19
20 # Evaluate KDE on the grid
21 density =

```

```
1 # Plot KDE result
2 fig, ax = plt.subplots(1, 2)
3
4 # Plot the KDE result
5 ax[0].plot(x, y, 'r')
6 ax[0].text(0.5, 0.5, 'Density Estimation')
7
8 # Plot the KDE result
9 ax[1].plot(x, y, 'r')
10 ax[1].text(0.5, 0.5, 'Density Estimation')
```



The resulting image from the Kernel Density Estimation (KDE) highlights the regions in Africa with the highest population densities. The darkest red areas represent the highest densities of population, indicating hotspots where the concentration of people is greatest. Conversely, lighter red areas show lower population densities. This visualization helps identify regions with varying

population intensities, which can be useful for various applications such as resource allocation, urban planning, and understanding demographic distributions. The possible mismatches between the resulting maps of the KDE-based estimation and Hotspot mostly be accounted for by the continuous-space approximation applied here versus the country-level spatial units used before.

100. Interpolation Methods - Inverse Distance Weighting

Inverse Distance Weighting (IDW) is a popular spatial interpolation method used to estimate unknown values at specific locations based on known values at surrounding points. The fundamental idea behind IDW is that points closer to the location of interest have a greater influence on the interpolated value than points farther away. This also relates to the famous idiom, Tobler's first law of geography, 'Everything is related to everything else, but near things are more related than distant things'.

We start by importing necessary libraries, preparing the dataset, and conducting data preparation steps using Africa as the usual example. Then, we compute the population density values for each country and introduce NaN values in the population density column for a few randomly selected countries. This will simulate the real-world scenario where some data might be unavailable; hence, we need to estimate their values using analytical techniques. We present a side-by-side comparison of map visuals to highlight the missing values.

Then, we present an IDW approach to approximate the values of these missing numbers. We define a function `idw_interpolation` that uses the `cKDTree` algorithm from the `scipy.spatial` module to find the k-nearest neighbors for each point where the data is missing. The function calculates weights based on the inverse distance, giving higher weights to closer points and normalizing these weights. It then computes the interpolated values as a weighted average of the known points.

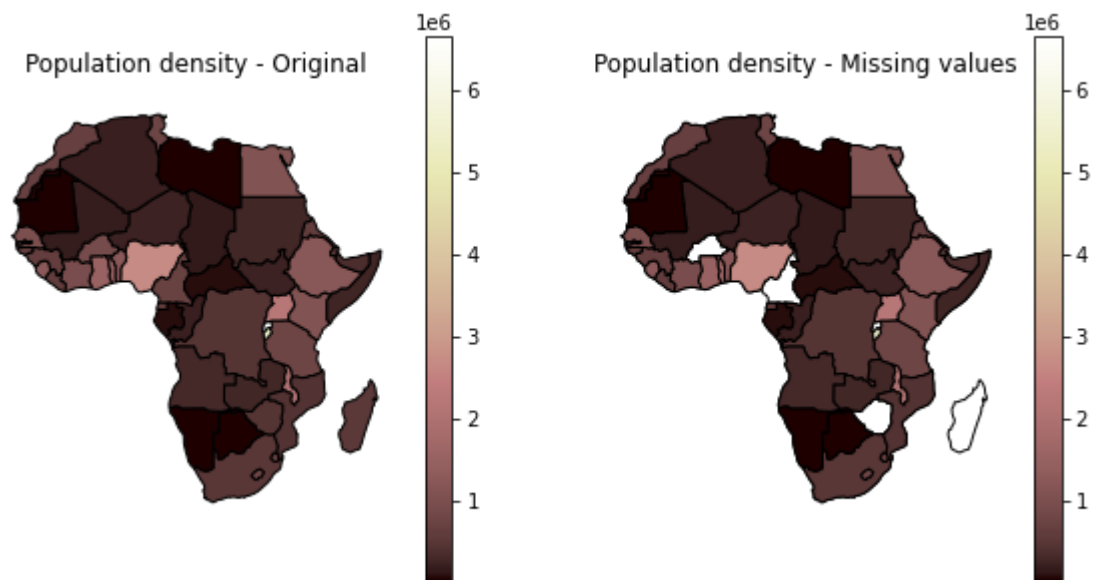
Finally, we wrap up the results. We prepare the data by separating the known and unknown values and extracting their coordinates and

population densities. After performing the IDW interpolation, the interpolated values are assigned back to the GeoDataFrame. We visualize the results in a side-by-side plot comparing the original and the interpolated maps and present a correlation analysis between the original and interpolated values.

In

```
1 # Library import
2 import geopandas as gpd
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # Load world countries dataset
7 gdf =
8
9 # Filter for Africa
10 gdf_africa = gdf[gdf['continent'] == 'Africa']
11
12 gdf_africa['area'] =
13 gdf_africa['pop_density'] = gdf_africa['pop_est'] / gdf_africa['area']
14 gdf_africa_missing =
15
16
17 # Simulating missing data, with 4 randomly picked countries
18 # By introduce missing values
19 indices_to_replace = [78, 48, 65])
20
21
22 # Show missing data
```

```
23 vmin =
24 vmax =
25
26 f, ax = 2, 5))
27
28
29
30
31
32
33
34
35
36 density - Original",
37         fontsize = 12, pad = 12)
38 density - Missing values",
39         fontsize = 12, pad = 12)
40
41 for aax in ax:
42
```



Next, we perform the IDW interpolation to estimate the missing population density values.

In

```
1 # Importing scipy
2 from scipy.spatial import cKDTree
3
4 # Defining a function to perform the IDW
5 def idw_interpolation(xi, yi, zi, xi_interp, yi_interp,
6     """
7     Perform IDW interpolation.
8     xi, yi: Coordinates of known points
9     zi: Values of known points
10     xi_interp, yi_interp: Coordinates of points to interpolate
11     power: Inverse distance power
12     """
13     tree = cKDTree([xi, yi])
14     # k nearest neighbors
```

```

15     distances, idx = yi_interp],
16     weights = 1 /
17     weights /=
18     zi_interp = * zi[idx],
19     return zi_interp
20
21 # Prepare data for interpolation
22 gdf_africa_interpol =
23 known =
24 unknown =
25
26 xi =
27 yi =
28 zi =
29
30 xi_interp =
31 yi_interp =
32
33 # Perform IDW interpolation
34 zi_interp = idw_interpolation(xi, yi, zi, xi_interp, yi_interp)
35
36 # Assign interpolated values back to the GeoDataFrame
37
38         'pop_density'] = zi_interp

```

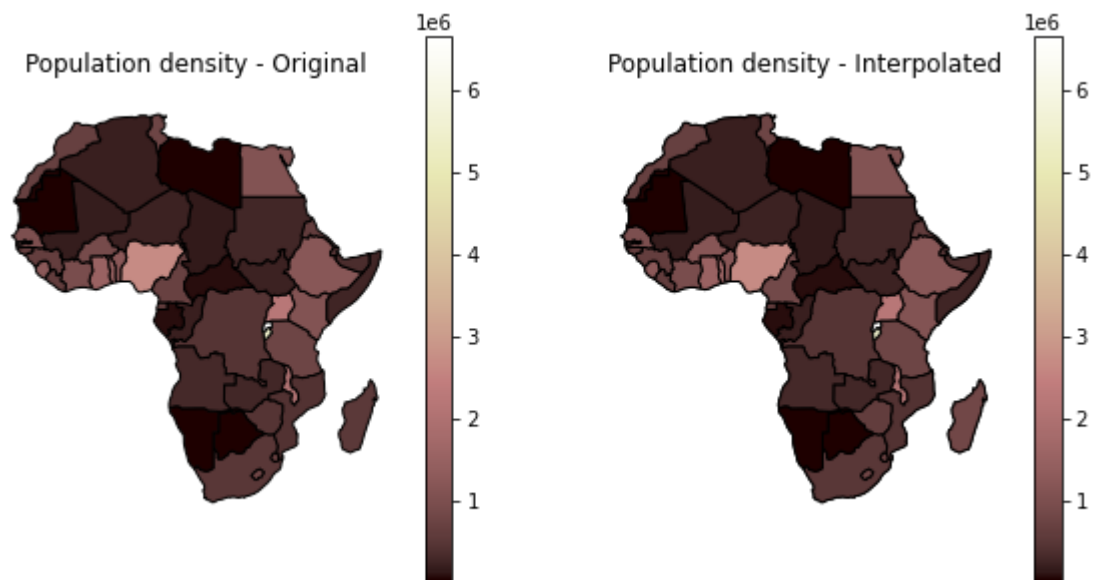
In

```

1 # Plot the results

```

```
2 f, ax = 2, 5))
3
4
5
6
7
8
9
10
11
12
13
14
15 density - Original", \
16         fontsize = 12, pad = 12)
17 density - Interpolated", \
18         fontsize = 12, pad = 12)
19
20 for aax in ax:
21
```



In

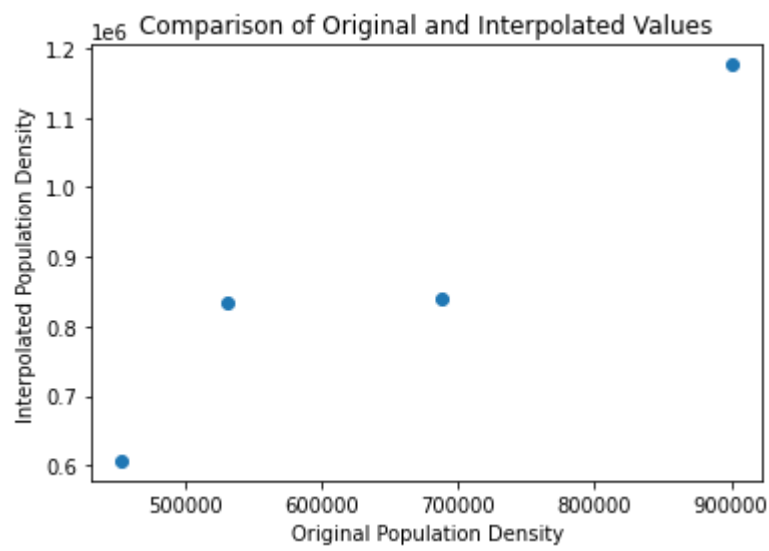
```

1 # Compare original and interpolated values for missing countries
2 missing_countries = \
3
4
5
6 missing_original = \
7     \
8
9
10 missing_interpol = \
11
12     [['name', \
13
14
15 interpol_comparison =
16

```

17
18
19
20
21
22 'o')
23 Population Density")
24 Population Density")
25 of Original and Interpolated Values")
26

| | pop_density | pop_density_interpol |
|----------------------|-------------|----------------------|
| pop_density | 1.000000 | 0.946681 |
| pop_density_interpol | 0.946681 | 1.000000 |



In this example, we successfully applied Inverse Distance Weighting (IDW) to interpolate missing population density values for African countries. Both the side-by-side map comparisons and the correlation

analysis with a significantly high correlation value support the fact that our interpolation performed well, estimating the missing values fairly compared to their original values.

101. Spatial Clustering

Spatial clustering is a method used to identify groups of similar observations within spatial data. This technique is particularly useful in various fields such as environmental science, urban planning, and epidemiology, where it helps in identifying patterns and clusters in spatial datasets. In this example, we will use two popular clustering algorithms, DBSCAN and K-Means, to cluster the African countries based on their geographic centroids.

[DBSCAN](#) (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that identifies clusters based on the density of points in a region. It requires two parameters: epsilon (ϵ), which defines the maximum distance between two points to be considered as neighbors, and the minimum number of points required to form a dense region. DBSCAN is effective at identifying clusters of varying shapes and sizes, and it can also handle noise and outliers by classifying them as points not belonging to any cluster.

[K-Means](#) on the other hand, is a centroid-based clustering algorithm that partitions the data into k clusters, where each cluster is represented by the mean (centroid) of its points. The algorithm works iteratively to assign each point to the nearest cluster centroid and then recalculates the centroids based on the updated cluster memberships. K-Means is efficient for large datasets and is best suited for well-separated, spherical clusters.

In the first block, we import the necessary libraries, including DBSCAN and KMeans from `sklearn.cluster` and `matplotlib.pyplot` for plotting. We also import `numpy` for numerical operations and `geopandas` for handling spatial data. We will again use the initially prepared country-level data set of Africa. Additionally, we

compute the centroids of these countries and extract their x and y coordinates. These coordinates serve as the input for our clustering algorithms.

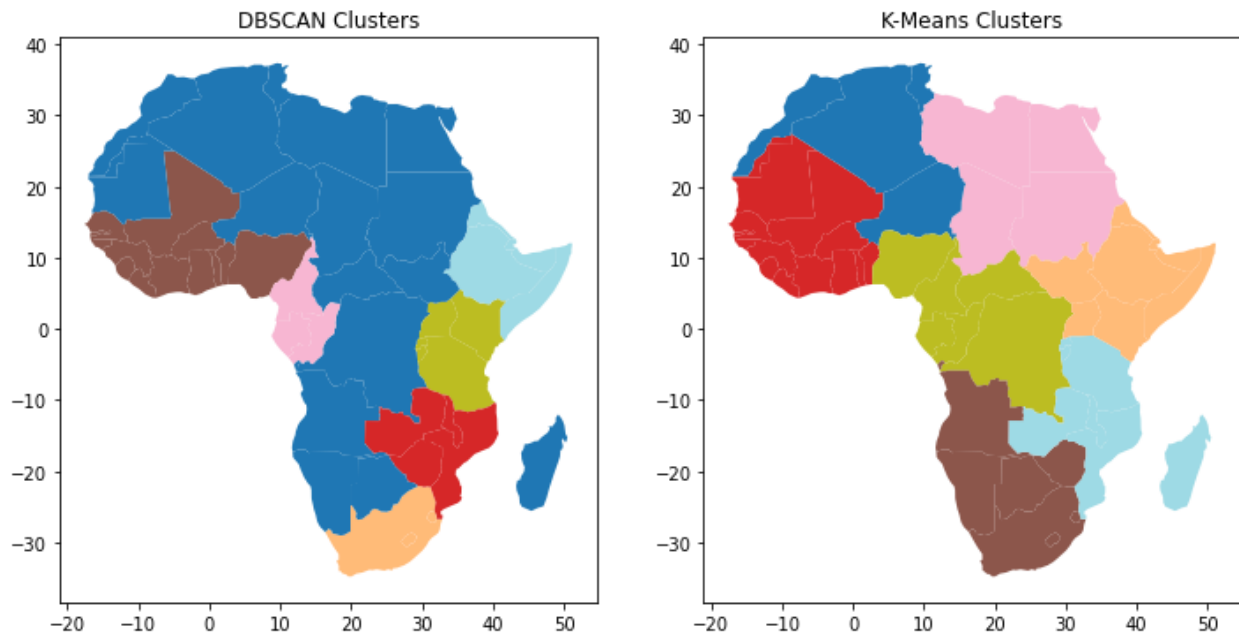
We apply the DBSCAN algorithm with an epsilon value of 6 and a minimum sample size of 3. The resulting cluster labels are added to our GeoDataFrame. Next, we apply the K-Means clustering algorithm, using the number of clusters identified by DBSCAN as the number of clusters for K-Means. The K-Means cluster labels have also been added to our GeoDataFrame.

Finally, we create a plot with two subplots, one for each clustering method. We use the plot method of GeoDataFrame to visualize the clusters on the map. The cmap parameter specifies the color map, and the set_title method sets the title for each subplot. The resulting plots show the clusters identified by DBSCAN and K-Means, helping us visualize the spatial distribution of the clusters.

In

```
1 # Import necessary libraries
2 from sklearn.cluster import DBSCAN, KMeans
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import geopandas as gpd
6
7 # Load world countries dataset
8 gdf =
9
10 # Filter for Africa
11
12
13 # Extract coordinates of centroids for clustering
14 gdf_africa['centroid_x'] =
```

```
15 gdf_africa['centroid_y'] =  
16 coords =  
17  
18 # DBSCAN Clustering  
19 db =  
20 gdf_africa['dbscan_cluster'] =  
21  
22 # K-Means Clustering  
23 kmeans =  
24  
25 gdf_africa['kmeans_cluster'] =  
26  
27 # Plot the clusters  
28 fig, ax = 2, 6))  
29  
30  
31 Clusters')  
32 Clusters')  
33
```



The DBSCAN clustering result on the left shows several small, distinct clusters and a significant portion of countries assigned to the same cluster, indicating that many countries are spatially similar based on the chosen parameters. In contrast, the K-Means clustering result on the right displays a more even distribution of clusters across Africa, with each cluster covering larger, contiguous regions, suggesting that K-Means identifies broader regional similarities compared to DBSCAN's more localized clustering. Comparative maps like these help us pick the right clustering method depending on the characteristics of the input data and the specific use case.

Summary on Spatial Statistics and ML

In this chapter, we covered basic statistics as well as a range of spatial regression models and machine learning techniques tailored for geospatial data analysis. We began with descriptive statistics to understand the fundamental characteristics of our spatial data, followed by methods to assess global and local spatial autocorrelation using Moran's I and Local Moran's I statistics. These techniques helped us identify clustering and dispersion patterns within our data.

We then demonstrated spatial feature generation, creating new attributes based on spatial relationships, which were used in subsequent regression analyses. The Ordinary Least Squares (OLS) regression model was introduced, providing a baseline for spatial data regression. We further explored advanced spatial regression models, including the Spatial Lag Model (SLM), Spatial Error Model (SEM), and Random Forest, to account for spatial dependencies and autocorrelation. The chapter also compared the goodness-of-fit metrics for the different regression models, guiding the selection of the most appropriate model for different spatial data scenarios.

Then, we explored hotspot analysis with the Getis-Ord G_i^* statistic and Kernel Density Estimation (KDE) while learning about Inverse Distance Weighting (IDW), which provides a robust method for spatial interpolation. In addition, we introduced spatial clustering techniques using DBSCAN and K-Means algorithms, which helped identify clusters of similar observations within the spatial data.

By mastering these techniques, we are now equipped to handle complex spatial datasets, apply advanced spatial regression models, and leverage machine learning algorithms for insightful geospatial analysis and prediction.

Summary

The purpose of this book was to create a practical, no-nonsense guide to geospatial data science in Python, equipping you with essential skills and knowledge to tackle real-world projects. I aimed for this to be a Python cookbook with exactly 101 practical tips and tricks covered.

Writing this book has been quite a journey of consolidating years of hands-on experience in academia, the start-up world, and consulting projects — now wrapped up and shared with you. I hope it inspires you to dive deeper into geospatial data science and make impactful contributions through your upcoming projects.

Throughout the book, I covered a wide range of topics organized into ten chapters and 101 sections, with the following major milestones:

understanding how to connect geographic locations to data records, using Pandas and GeoPandas for effective data manipulation and visualization, core areas and techniques of spatial analytics like geocoding and spatial indexing, exploring both vector and raster data, and diving into advanced topics like spatial networks and geospatial statistics combined with machine learning.

Across the sections, I demonstrated the practical applications of these concepts through hands-on examples, focusing on how to use Python for geospatial analytics in various scenarios, from urban planning to transportation. Additionally, I shared numerous tips and best practices to help you write efficient code, perform accurate analyses, and create meaningful visualizations—from reading a data table to displaying the final maps.

To continue your learning journey, consider exploring further resources such as more advanced books (for that, take a look at my online courses, including mine, and make sure to join my geospatial data science community on

Now that you have the foundational knowledge and skills, put it into practice. Start a new project, explore new datasets, and share your findings. Your journey in geospatial data science is just beginning!

Budapest, 2024

Milan Janosov